# МЕТОДЫ ВЕРИФИКАЦИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

**В.В. Кулямин**

Институт системного программирования РАН
109004, г. Москва, ул. Б. Коммунистическая, д. 25

**Аннотация.** В данной работе представлен обзор методов верификации программного обеспечения (ПО). Верификацией называется проверка соответствия результатов отдельных этапов разработки программной системы требованиям и ограничениям, сформулированным для них на предыдущих этапах. Рассматривается контекст использования различных методов контроля качества и, в частности, верификации в рамках жизненного цикла ПО. Представлено содержание международных стандартов ISO и IEEE, регулирующих проведение различных видов верификации при разработке программ. Предлагается классификация известных методов верификации, полученная обобщением сложившейся практики позиционирования работ в этой области. В рамках предложенной системы рассматриваются как зрелые и широко применяемые при промышленной разработке методы верификации ПО, так и недавно созданные и используемые пока только в рамках исследовательских проектов. Обзор включает методы статического анализа программ, методы проведения инспекций и экспертиз, техники анализа архитектуры ПО, методы проверки моделей и дедуктивного анализа корректности ПО, а также методы динамической верификации — тестирование и верификационный мониторинг. Рассматриваются инструменты поддержки верификации ПО.

**Annotation.** The paper presents review of software verification methods. By verification we mean conformance checking of results of separate phases of software development to requirements and restrictions stated for these results on previous phases. The article considers use of quality control methods, including verification methods, in software development lifecycle. It also presents contents of international standards of ISO and IEEE

regulating verification activities in software development. The paper gives classification of verification methods based on current practice of research work positioning in this area. This classification used to review both mature verification methods, widely used in industrial software development, and research ones developed only recently. The review includes methods of static analysis, software review and inspection techniques, methods of software architecture analysis, model checking and theorem proving, along with dynamic verification methods — testing and monitoring. Tool support for each considered method is also discussed.

# Содержание


Введение ........ 4
1. Основные понятия ........ 6
  1.1. Верификация и валидация ........ 7
  1.2. Характеристики качества программного обеспечения ........ 9
2. Место верификации в жизненном цикле ПО ........ 15
  2.1. Задачи верификации в рамках жизненного цикла ПО ........ 16
  2.2. Верификация и другие процессы разработки и сопровождения ПО ........ 17
  2.3. Верификация различных артефактов жизненного цикла ПО ........ 18
  2.4. Международные стандарты, касающиеся верификации ПО ........ 23
3. Методы верификации программного обеспечения ........ 29
  3.1. Экспертиза ........ 34
    3.1.1. Оценка ПО по Фагану ........ 35
    3.1.2. Другие виды общих экспертиз ........ 38
    3.1.3. Специализированные методы экспертиз ........ 41
    3.1.4. Методы анализа архитектуры ПО ........ 43
  3.2. Статический анализ ........ 46
  3.3. Формальные методы верификации ........ 47
    3.3.1. Логико-алгебраические модели ........ 47
    3.3.2. Исполнимые модели ........ 51
    3.3.3. Модели промежуточного типа ........ 57
    3.3.4. Классификация формальных методов ........ 59
    3.3.5. Методы и инструменты дедуктивного анализа ........ 60
    3.3.6. Методы и инструменты проверки моделей ........ 63
    3.3.7. Методы и инструменты проверки согласованности ........ 65
  3.4. Динамические методы верификации ........ 66
    3.4.1. Мониторинг ........ 69
    3.4.2. Тестирование ........ 72
    3.4.3. Виды тестирования ........ 75
    3.4.4. Критерии полноты тестирования ........ 78
    3.4.5. Техники построения тестов ........ 80
    3.4.6. Инструменты автоматизации тестирования ........ 86
  3.5. Синтетические методы ........ 88
    3.5.1. Тестирование на основе моделей ........ 88
    3.5.2. Мониторинг формальных свойств ПО ........ 91
    3.5.3. Статический анализ формальных свойств ........ 92
    3.5.4. Синтетические методы генерации структурных тестов ........ 94
Заключение ........ 96
Литература ........ 97

## Введение

Информационные технологии являются одним из основных элементов инфраструктуры современного общества. Они служат базой для экономической деятельности и социального и культурного развития человечества, обеспечивая людям доступ к огромным массивам разнообразной информации и связывая их друг с другом, где бы они не находились.

Любая информационная система состоит из аппаратного и программного обеспечения (ПО). В начале развития компьютерной техники аппаратная часть была более сложной и значительно более дорогостоящей, стоимость программной части оценивалась примерно в 5% стоимости всей системы. Однако гибкость программного обеспечения и (как оказалось впоследствии, обманчивая) простота внесения в него изменений побуждали использовать его для решения разнообразнейших задач на одном и том же или стандартизированном аппаратном обеспечении. Поэтому постепенно ПО усложнялось, приобретало все большую ценность, и в последние десятилетия его стоимость достигает от 30% до 90% стоимости систем, в зависимости от их типа [1]. Совокупные затраты на создание, развитие и поддержку ПО уже превосходят соответствующие затраты на аппаратное обеспечение [2]. Сложность же современных программных комплексов такова, что многие исследователи считают их самыми сложными системами, созданными человеком [3].

Возрастающая сложность ПО приводит к увеличению количества ошибок в нем, а одновременный рост количества и критичности выполняемых им функций влечет рост ущерба от этих ошибок. Оценки потерь одной экономики США от некачественного программного обеспечения дают около 60 миллиардов долларов в год [4]. Известны также примеры серьезных ошибок в ПО, приведших к потере человеческих жизней, космических аппаратов или к масштабным нарушениям работы инфраструктурных сетей [5-11]. Одна из первых хорошо описанных ошибок такого рода — ошибка в системе управления космическим аппаратом Mariner 1 [5], которая привела к потере этого аппарата 22 июля 1962 года. Ошибка заключалась в том, что в одном месте была пропущена операция усреднения скорости корабля по нескольким последовательно измеренным значениям. В результате колебания значения скорости,

вызванные ошибками измерений, стали рассматриваться системой как реальные, и она попыталась предпринять корректирующие действия, которые привели к полной неуправляемости аппарата. Именно после этого инцидента управление военно-воздушных сил США приняло решение использовать в процессе разработки ПО экпертизу кода — его просмотр и анализ другими людьми, помимо самого разработчика.

При построении систем определенного уровня сложности люди в принципе не могут избежать ошибок, просто потому, что им вообще свойственно ошибаться, а возрастающая сложность предоставляет все больше возможностей для ошибок, при этом затрудняя их быстрое обнаружение. Для обеспечения корректности и надежности работы таких систем большое значение имеют различные методы верификации и валидации, позволяющие выявлять ошибки на разных этапах разработки и сопровождения ПО, чтобы последовательно устранять их.

Цель данной работы — представить обзор разнообразных методов верификации ПО. Но прежде, чем перейти к самому обзору, необходимо напомнить определения основных используемых понятий и определить место верификации среди других видов деятельности, используемых при разработке и сопровождении ПО.

## *1. Основные понятия*

Под *жизненным циклом программного обеспечения* обычно понимают весь интервал времени от момента зарождения идеи о том, чтобы создать или приобрести программную систему для решения определенных задач, до момента полного прекращения использования последней ее версии. Жизненным циклом этот период назван по аналогии с циклом жизни растения или животного, которое рождается, проходит определенные фазы роста и развития и в итоге погибает, давая жизнь новым существам, проходящим через те же стадии.

Описать общую структуру жизненного цикла произвольной программной системы, по-видимому, невозможно — слишком сильно отличается разработка и развитие ПО, предназначенного для решения разных задач в различных окружениях. Однако можно определить набор понятий, в терминах которых описывается любая такая структура — это, прежде всего, виды деятельности, роли и артефакты.

*Вид деятельности* в жизненном цикле ПО — это набор действий, направленных на решение одной задачи или группы тесно связанных задач в рамках разработки и сопровождения ПО. Примерами видов деятельности являются анализ предметной области, выделение и описание требований, проектирование, разработка кода, тестирование, управление конфигурациями, развертывание.

*Роль* в жизненном цикле ПО — это профессиональная специализация людей, участвующих в работах по созданию или сопровождению ПО (или затрагиваемых ими) и имеющих одинаковые интересы или решающих одни и те же задачи по отношению к этому ПО. Примеры ролей: бизнес-аналитик, инженер по требованиям, архитектор, проектировщик пользовательского интерфейса, программист-кодировщик, технический писатель, тестировщик, руководитель проекта, пользователь, администратор системы.

*Артефактами* жизненного цикла ПО называются различные информационные сущности, документы и модели, создаваемые или используемые в ходе разработки и сопровождения ПО. Так, артефактами являются техническое задание, описание архитектуры, модель предметной области на каком-либо графическом языке, исходный код, пользовательская документация и т.д. Различные модели, используемые

отдельными разработчиками при создании и анализе ПО, но не зафиксированные в виде доступных другим людям документов, не могут считаться артефактами.

## 1.1. Верификация и валидация

Верификация и валидация являются видами деятельности, направленными на контроль качества программного обеспечения и обнаружение ошибок в нем. Имея общую цель, они отличаются источниками проверяемых в их ходе свойств, правил и ограничений, нарушение которых считается ошибкой.

*Верификация* проверяет соответствие одних создаваемых в ходе разработки и сопровождения ПО артефактов другим, ранее созданным или используемым в качестве исходных данных, а также соответствие этих артефактов и процессов их разработки правилам и стандартам. В частности, верификация проверяет соответствие между нормами стандартов, описанием требований (техническим заданием) к ПО, проектными решениями, исходным кодом, пользовательской документацией и функционированием самого ПО. Кроме того, проверяется, что требования, проектные решения, документация и код оформлены в соответствии с нормами и стандартами, принятыми в данной стране, отрасли и организации при разработке ПО, а также — что при их создании выполнялись все указанные в стандартах операции, в нужной последовательности. Обнаруживаемые при верификации ошибки и дефекты являются расхождениями или противоречиями между несколькими из перечисленных документов, между документами и реальной работой программы, между нормами стандартов и реальным процессами разработки и сопровождения ПО. При этом принятие решения о том, какой именно документ подлежит исправлению (может быть, и оба) является отдельной задачей.

*Валидация* проверяет соответствие любых создаваемых или используемых в ходе разработки и сопровождения ПО артефактов нуждам и потребностям пользователей и заказчиков этого ПО, с учетом законов предметной области и ограничений контекста использования ПО. Эти нужды и потребности чаще всего не зафиксированы документально — при фиксации они превращаются в описание требований, один из артефактов процесса разработки ПО. Поэтому валидация является

менее формализованной деятельностью, чем верификация. Она всегда проводится с участием представителей заказчиков, пользователей, бизнес-аналитиков или экспертов в предметной области — тех, чье мнение можно считать достаточно хорошим выражением реальных нужд и потребностей пользователей, заказчиков и других заинтересованных лиц. Методы ее выполнения часто используют специфические техники выявления знаний и действительных потребностей участников.

Различие между верификацией и валидацией проиллюстрировано на Рис. 1.

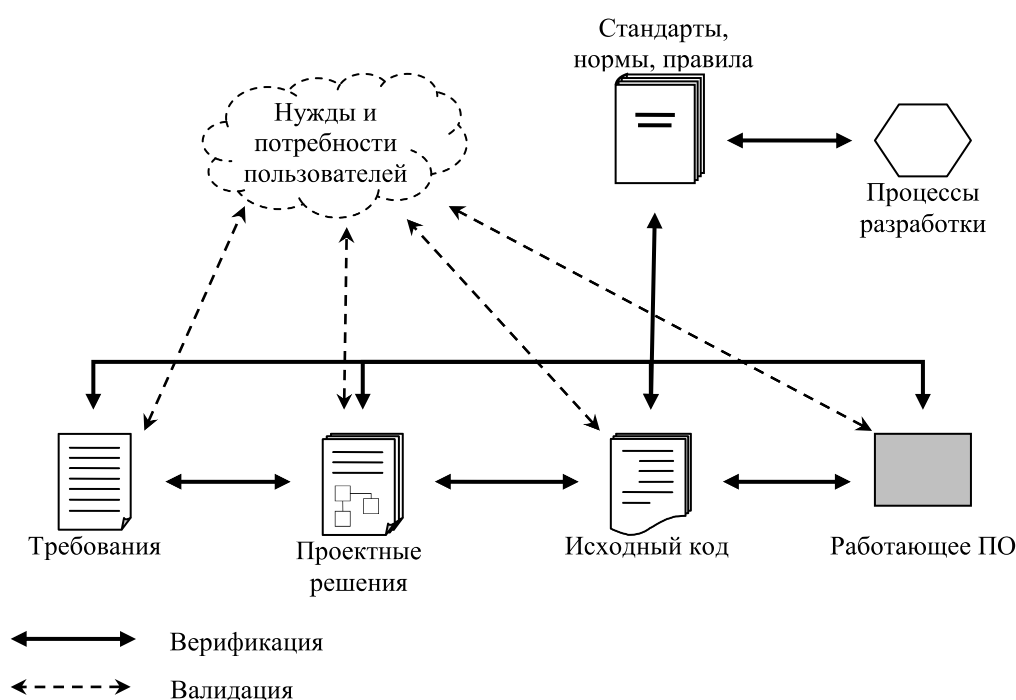


**Рисунок 1. Соотношение верификации и валидации.**

Приведенные определения получены некоторым расширением определений из стандарта IEEE 1012 на процессы верификации и валидации [12]. В стандартном словаре терминов программной инженерии IEEE 610.12 1990 года [13] определение верификации по смыслу примерно то же, а определение валидации несколько другое — там говорится, что валидация должна проверять соответствие полученного в результате разработки ПО исходным требованиям к нему. В этом случае валидация являлась бы частным случаем верификации, что нигде в литературе по программной инженерии не отмечается, поэтому, а также потому, что оно поправлено в IEEE 1012 2004 года, это

определение следует считать неточным. Частое использование фразы B. Boehm'а [14]: «Верификация отвечает на вопрос "Делаем ли мы продукт правильно?", а валидация — на вопрос "Делаем ли мы правильный продукт?"» также добавляет путаницы, поскольку афористичность этого высказывания, к сожалению, сочетается с двусмысленностью. Однако многочисленные труды его автора позволяют считать, что он подразумевал под верификацией и валидацией примерно те же понятия, которые определены выше.

Указанные разночтения можно проследить и в содержании стандартов программной инженерии. Так, стандарт ISO 12207 [15] считает тестирование разновидностью валидации, но не верификации, что, по-видимому, является следствием использования неточного определения из стандартного словаря [13].

В среде исследователей, занимающихся теоретической информатикой (computer science), широко распространено более узкое понимание термина «верификация» — только как формальной верификации. В данном обзоре мы будем пользоваться более широким, инженерным понятием, хотя различные методы формальной верификации тоже будут рассмотрены.

## 1.2. Характеристики качества программного обеспечения

Поскольку основной задачей верификации, как и валидации, является контроль *качества программного обеспечения*, необходимо обратиться к этому понятию. Наиболее широко используется определение качества ПО в виде системы атрибутов или факторов, которые могут быть оценены с помощью ряда метрик. Такой подход позволяет конструктивно оценивать качество ПО в целом, во всех необходимых аспектах. Впервые он был сформулирован в 1977 году в работе МакКола с соавторами [16], затем несколько раз пересматривался в работах [17-20], и в итоге был принят в качестве стандартной модели качества ISO 9126 [21] в 1991 году. Ниже описана модель качества ПО, описанная в действующей на сегодняшний день версии этого стандарта, принятой в 2001 году. В ходе идущей переработки этого и других связанных с оценкой качества ПО стандартов по новой схеме SQauRE [25] серьезные изменения в модели качества не планируются.

**Рисунок 2. Факторы и атрибуты внешнего и внутреннего качества ПО по ISO 9126.**

Стандарт ISO 9126 [21-24] предлагает учитывать три разных точки зрения при рассмотрении качества ПО: точку зрения разработчиков, которые воспринимают *внутреннее качество ПО*, точку зрения руководства и аттестации ПО на соответствие сформулированным к нему требованиям, в ходе которой определяется *внешнее качество ПО*, и точку зрения пользователей, ощущающих *качество ПО при использовании*. Во всех трех случаях для описания качества используется предложенная МакКолом многоуровневая модель, состоящая из целей или факторов, атрибутов или критериев и метрик качества. Цели (факторы) позволяют на верхнем уровне определять основные характеристики, которые ПО должно иметь или уже имеет. Каждый фактор состоит из набора атрибутов (критериев), позволяющих качественно описать желаемые или полученные характеристики более детально. Каждый атрибут поддерживается набором метрик, которые позволяют количественно оценивать наличие соответствующей характеристики.

Для двух точек зрения — внешнего качества и внутреннего качества — в рамках ISO 9126 предложена модель качества, состоящая из 6 факторов и 27 атрибутов и схематически представленная на Рис. 2.

Определения факторов и атрибутов качества в этой модели приведены ниже.

- Функциональность (functionality) — способность ПО в определенных условиях решать задачи, нужные пользователям. Определяет, что именно делает ПО.
    - Функциональная пригодность (suitability) — способность решать нужный набор задач.
    - Точность (accuracy) — способность выдавать нужные результаты.
    - Способность к взаимодействию, совместимость (interoperability) — способность взаимодействовать с нужным набором других систем.
    - Соответствие стандартам и правилам (compliance) — соответствие ПО имеющимся стандартам, нормативным и законодательным актам, другим регулирующим нормам.
    - Защищенность (security) — способность предотвращать неавторизованный и не разрешенный доступ к данным, коммуникациям и др. элементам ПО.
- Надежность (reliability) — способность ПО поддерживать определенную работоспособность в заданных условиях.
    - Зрелость, завершенность (maturity) — величина, обратная частоте отказов ПО. Определяется средним временем работы без сбоев и величиной, обратной вероятности возникновения отказа за данный период времени.
    - Устойчивость к отказам (fault tolerance) — способность поддерживать заданный уровень работоспособности при отказах и нарушениях правил взаимодействия с окружением.
    - Способность к восстановлению (recoverability) — способность восстанавливать определенный уровень работоспособности и целостность данных после отказа в рамках заданных времени и ресурсов.
    - Соответствие стандартам надежности (reliability compliance).
- Удобство использования (usability) или практичность — способность ПО быть удобным в обучении и использовании, а также привлекательным для пользователей.
    - Понятность (understandability) — показатель, обратный к усилиям, которые затрачиваются пользователями на восприятие основных понятий ПО и осознание способов их использования для решения своих задач.

- Удобство обучения (learnability) — показатель, обратный к усилиям, затрачиваемым пользователями на обучение работе с ПО.
- Удобство работы (operability) — показатель, обратный трудоемкости решения пользователями задач с помощью ПО.
- Привлекательность (attractiveness) — способность ПО быть привлекательным для пользователей.
- Соответствие стандартам удобства использования (usability compliance).

- Производительность (efficiency) или эффективность — способность ПО при заданных условиях обеспечивать необходимую работоспособность по отношению к выделяемым для этого ресурсам.
  - Временная эффективность (time behaviour) — способность ПО решать определенные задачи за отведенное время.
  - Эффективность использования ресурсов (resource utilisation) — способность решать нужные задачи с использованием заданных объемов ресурсов определенных видов. Имеются в виду такие ресурсы, как оперативная и долговременная память, сетевые соединения, устройства ввода и вывода, и пр.
  - Соответствие стандартам производительности (efficiency compliance).
- Удобство сопровождения (maintainability) — удобство проведения всех видов деятельности, связанных с сопровождение программ.
  - Анализируемость (analyzability) или удобство проведения анализа — удобство проведения анализа ошибок, дефектов и недостатков, а также удобство анализа необходимости изменений и их возможных последствий.
  - Удобство внесения изменений (changeability) — показатель, обратный трудозатратам на выполнение необходимых изменений.
  - Стабильность (stability) — показатель, обратный риску возникновения неожиданных эффектов при внесении необходимых изменений.
  - Удобство проверки (testability) — показатель, обратный трудозатратам на проведение тестирования и других видов проверки того, что внесенные изменения привели к нужным результатам.

    - Соответствие стандартам удобства сопровождения (maintainability compliance).
- Переносимость (portability) — способность ПО сохранять работоспособность при переносе из одного окружения в другое, включая организационные, аппаратные и программные аспекты окружения.

  Иногда в русскоязычной литературе эта характеристика называется мобильностью. Однако термин «мобильность» стоит зарезервировать для перевода «mobility» — способности ПО и системы в целом сохранять работоспособность при ее физическом перемещении в пространстве.
    - Адаптируемость (adaptability) — способность ПО приспосабливаться к различным окружениям без проведения для этого действий, помимо заранее предусмотренных.
    - Удобство установки (installability) — способность ПО быть установленным или развернутым в определенном окружении.
    - Способность к сосуществованию (coexistence) — способность ПО сосуществовать в общем окружении с другими программами, деля с ними ресурсы.
    - Удобство замены (replaceability) другого ПО данным — возможность применения данного ПО вместо других программных систем для решения тех же задач в определенном окружении.
    - Соответствие стандартам переносимости (portability compliance).

Указанные выше характеристики используются для описания качества ПО с точки зрения его разработчиков и их руководства. Для пользовательской точки зрения, т.е. качества ПО при использовании, стандарт ISO 9126 [21] предлагает другую систему факторов.

- Эффективность (effectiveness) — способность решать задачи пользователей с необходимой точностью при использовании в заданном контексте.
- Продуктивность (productivity) — способность предоставлять определенные результаты в рамках ожидаемых затрат ресурсов.

- Безопасность (safety) — способность обеспечивать необходимо низкий уровень риска нанесения ущерба жизни и здоровью людей, бизнесу, собственности или окружающей среде.
- Удовлетворение пользователей (satisfaction) — способность приносить удовлетворение пользователям при использовании в заданном контексте.

В дальнейшем мы будем использовать систему из шести факторов, предназначенную для оценки и описания внешнего или внутреннего качества ПО, поскольку все методы верификации затрагивают именно эти его аспекты.

## *2. Место верификации в жизненном цикле ПО*

Хотя общую структуру жизненного цикла произвольной программной системы определить невозможно, существует несколько наиболее часто используемых способов организации различных видов деятельности в рамках жизненного цикла. Их называют моделями жизненного цикла ПО. Чаще всего работы организуются либо в соответствии с *каскадной (*или *водопадной) моделью* [26,27] (см. Рис. 3, слева), либо в рамках одной из многочисленных разновидностей *итеративной модель* жизненного цикла, впервые описанной в 1970 году [27] (Рис. 3, справа).

**Рисунок 3. Схема каскадной и итеративной моделей жизненного цикла ПО.**

Каскадная модель хорошо работает в тех случаях, когда требования к создаваемой системе удается полностью выявить и зафиксировать в начале проекта (что на практике случается не часто), и результаты всех выполняемых действий проходят тщательный анализ на внутреннюю корректность и соответствие исходным данным. В противном случае обнаруживаемые впоследствии ошибки и недоработки в результатах предыдущих шагов существенно затрудняют продвижение проекта и снижают его управляемость. Таким образом, в рамках каскадной модели верификация имеет должна выполняться в рамках всех видов деятельности для проверки

корректности их результатов, и именно она в первую очередь обеспечивает успешное движение к конечной цели. Один из видов верификации — тестирование — даже выделяется в отдельный этап проекта.

В рамках итеративной модели отдельные виды деятельности уже не привязаны к этапам проекта и могут выполняться в разнообразных комбинациях. Итеративная модель позволяет быстро реагировать на изменения требований, но требует большего умения от руководителя проекта. В ее рамках различные методы верификации также имеют важнейшее значение, поскольку только с их помощью можно получить оценку качества результатов проекта, как конечных, так и промежуточных. Именно оценка качества служит основной информацией для оценки продвижения к целям проекта, планирования следующих итераций, принятия решений о прекращении проекта или передаче его результатов заказчику.

## 2.1. Задачи верификации в рамках жизненного цикла ПО

Все используемые на практике модели жизненного цикла по схеме организации работ являются разновидностями либо каскадной, либо итеративной модели, поэтому независимо от процесса разработки ПО верификация играет в нем ключевую роль, решая следующие задачи.

- Выявление дефектов (ошибок, недоработок, неполноты и пр.) различных артефактов разработки ПО (требований, проектных решений, документации или кода), что позволяет устранять их и поставлять пользователям и заказчикам более правильное и надежное ПО.
- Выявление наиболее критичных и наиболее подверженных ошибкам частей создаваемой или сопровождаемой системы.
- Контроль и оценка качества ПО во всех его аспектах.
- Предоставление всем заинтересованным лицам (руководителям, заказчикам, пользователям и пр.) информации о текущем состоянии проекта и характеристиках его результатов.

- Предоставление руководству проекта и разработчикам информации для планирования дальнейших работ, а также для принятия решений о продолжении проекта, его прекращении или передаче результатов заказчику.

## 2.2. Верификация и другие процессы разработки и сопровождения ПО

*Процессом жизненного цикла ПО* называется группа видов деятельности, выполняемых для решения определенного набора связанных задач по разработке или сопровождению ПО. На сегодняшний день нет четко определенного общего списка процессов, который не вызывал бы возражений у тех или иных исследователей и практиков. Международные стандарты ISO 12207 [15], IEEE 1074 [28], ISO 15288 [29], ISO 15504 [30] используют несколько отличающиеся системы процессов.

- По ISO 12207 к верификации имеют отношение 5 процессов: обеспечение качества (quality assurance), собственно верификация, валидация, совместные экспертизы (joint review) и аудит (audit). Тестирование целиком отнесено к валидации. Кроме того, выделен процесс разрешения проблем (problem resolution), для которого верификация и валидация поставляют входные данные (те самые проблемы).
- IEEE 1074 выделяет только один связанный с верификацией процесс — группу деятельностей по оценке (evaluation), которая включает экспертизы (review), аудиты, прослеживание требований и тестирование. Еще несколько видов деятельности, которые можно отнести к верификации и валидации, разбросаны по другим процессам — сбор и анализ метрик, анализ осуществимости, определение потребностей в улучшении ПО, валидация программы обучения.
- ISO 15288 считает отдельными процессами управление качеством, оценивание, верификацию и валидацию.
- В ISO 15504 (SPICE) в качестве процессов выделены совместные экспертизы и аудиты (один процесс), управление качеством, обеспечение качества и экспертизы (review). Тестирование считается частью других процессов —

реализации и интеграции ПО и интеграции программно-аппаратной системы в целом.

С технической точки зрения верификация и валидация являются неотъемлемыми элементами деятельности по обеспечению качества. С одной стороны, эта деятельность должна обеспечить формирование критериев качества, использование при разработке доказавших свою эффективность технологий, определение и контроль процедур выполнения отдельных операций, точность, согласованность и полноту при описании требований, проектных решений, пользовательской документации, формулировку требований и проектных решений на необходимом уровне абстракции. С другой стороны, в рамках обеспечения качества с помощью верификации и валидации необходимо оценивать текущие характеристики качества ПО и отдельных артефактов процесса разработки и сопоставлять их с критериями и правилами, определенными в рамках системы обеспечения качества проекта и организации в целом.

Деятельность по управлению качеством отличается от обеспечения качества, по-видимому, только большим акцентом на административных процедурах и предварительном определении целей обеспечения качества, на прямой ответственности руководства проекта за качество его результатов.

Экспертизы и аудит, в свою очередь, являются методами проведения верификации и валидации, такими же, как тестирование, оценка архитектуры на основе сценариев или проверка моделей. В стандартах они рассматриваются как отдельные процессы, скорее всего, потому что применимы к произвольным артефактам жизненного цикла в рамках любого вида деятельности, а также часто используются для оценки процессов и организационных видов деятельности в проекте, в отличие от большинства других методов верификации.

## 2.3. Верификация различных артефактов жизненного цикла ПО

Артефакты жизненного цикла ПО можно разделить на технические и организационные. К техническим артефактам относятся описание требований (техническое задание), описание проектных решений (эскизный и технический проекты), исходный код (текст программы), документация пользователей и

администраторов (рабочая документация), сама работающая система. Техническими также являются вспомогательные артефакты для проведения верификации и валидации — формальные модели требований и проектных решений, наборы тестов и компоненты тестового окружения, модели поведения реального окружения системы. Организационными артефактами являются структура работ, разнообразные проектные планы (план-график работ, план конфигурационного управления, план обеспечения качества, план обхода и преодоления рисков, планы проверок и испытаний и пр.), описания системы качества, описания процессов и процедур выполнения отдельных работ. Верификация может и должна проводиться для всех видов артефактов, создаваемых при разработке и сопровождении программных систем.

- При верификации организационных документов и процессов проверяется, насколько выбранные формы организации, планы и методы выполнения работ соответствуют задачам, решаемым в рамках проекта, и ограничениям по срокам и бюджету, то есть, что с помощью выбранных методов и технологий проект действительно можно выполнить в рамках контракта.

  Проверяется также, что команда проекта в достаточной степени владеет используемыми технологиями разработки, или же, что запланированы необходимые мероприятия по обучению.

  В дальнейшем в рамках данной статьи рассматриваются, в основном, методы верификации, нацеленные на оценку качества технических, а не организационных артефактов процесса разработки.

- При верификации описания требований одной из первых задач верификации является оценка осуществимости требований с помощью технологий, взятых на вооружение в проекте и в рамках выделенных на проект ресурсов. Проверяются также характеристики требований, указанные в стандартах IEEE 830 [31] и IEEE 1233 [32], а именно следующие.
  - Однозначность. Требования должны однозначно, недвусмысленно выражать нужные ограничения.
  - Непротиворечивость или согласованность. Различные требования не должны противоречить друг другу или основным законам предметной области.

    - Внутренняя полнота. Требования должны описывать поведения системы во всех возможных в контексте ее работы ситуациях. Все значимые законы предметной области и нормы действующих в ней стандартов должны быть учтены в требованиях как ограничения на работу системы.
    - Минимальность. Требования не должны быть сводимы друг к другу на основе формальной логики и основных законов предметной области.
    - Проверяемость. В каждой затрагиваемой требованием ситуации должен быть способ однозначно установить, выполнено оно или нарушено.
    - Систематичность. Требования должны быть представлены в рамках единой системы, с четким указанием связей между ними, с уникальными идентификаторами и набором определенных атрибутов: приоритетом, риском внесения изменений, критичностью для пользователей и пр.

  Кроме этого, требования должны адекватно и достаточно полно отражать нужды и потребности пользователей и других заинтересованных лиц. Требования должны затрагивать все существенные для пользователей аспекты качества системы: помимо функциональных требований, должны быть адекватно отражены требования к производительности, надежности, удобству использования, переносимости и удобству сопровождения. Для проверки адекватности и полноты отражения реальных потребностей пользователей необходимо проводить валидацию.

- При верификации проектных решений проверяются следующие свойства.
    - Все проектные решения связаны с требованиями и действительно нацелены на их реализацию. Все требования нашли отражение в проектных решениях.
    - Проектные документы точно и полно формулируют принятые решения, отдельные их элементы не противоречат друг другу.
    - При оформлении проектных документов учтены все правила корректности составления документов такого типа на соответствующих языках. Если используются графические нотации, такие как DFD, ERD или UML, то все диаграммы составлены с соблюдением всех правил и ограничений этих языков.

    - Для проектных решений, связанных с критически важными требованиями к системе, например, по ее безопасности и защищенности, необходимо с помощью максимально строгих методов установить их корректность, т.е. то, что они действительно реализуют соответствующие требования во всех возможных в контексте работы системы ситуациях.
- При верификации исходного кода системы проверяют указанные ниже характеристики.
    - Все элементы кода связаны с проектными решениями и требованиями и корректно реализуют соответствующие проектные решения.
    - Код написан в соответствии с синтаксическими и семантическими правилами выбранных языков программирования, а также с принятыми в организации и данном проекте стандартами оформления текстов программ (coding rules, coding conventions). Выполнены требования к удобству сопровождения кода, в коде отсутствуют неясные места, все его элементы можно протестировать с помощью сценариев возможной работы системы.
    - В исходном коде отсутствуют пути выполнения, достижимые в условиях работы системы и приводящие к ее сбоям, зацикливаниям или тупиковым ситуациям, разрушению процессов и данных проверяемой системы или объемлющей, исключительным ситуациям, непредусмотренным в требованиях и проектных решениях, и пр. Во всех возможных в контексте работы системы сценариях выполнения кода принятые проектные решения и требования соблюдаются, и нет элементов кода, выполняющих непредусмотренные требованиями действия. Эти правила на практике невозможно проверить полностью, но при верификации стремятся как можно более достоверно подтвердить его. При возрастании критичности требований, связанных с компонентами и элементами кода, требуется более строгое подтверждение, и используются более строгие и трудоемкие методы.
- Верификация самой работающей системы или ее компонентов, которые можно выполнять независимо, призвана проверить следующее.

  - Система или ее компоненты действительно способны работать в том окружении, в котором они нужны пользователям, или же в рамках достаточно точной имитации этого окружения.
  - Поведение системы или ее компонентов на возможных сценариях их использования соответствует требованиям по всем измеримым характеристикам. Это, снова, невозможно проверить полностью. Однако, для наиболее критичных требований и сценариев использования применяются более строгие и полные методы проверки соответствия.

  Часто проверяется также соответствие поведения системы и ее компонентов реальным нуждам пользователей — это уже является валидацией.

- При верификации пользовательской документации проверяется следующее.
  - Документация содержит полное, точное и непротиворечивое описание поведения системы.
  - Описанное в документации поведение соответствует реальному поведению системы.
- Верификации также должны подвергаться тестовые планы или планы других мероприятий по верификации, а также тесты или материалы, подготовленные для проведения верификации других артефактов, например различные формальные модели. В этих случаях проверяются такие характеристики.
  - Подготовленные планы соответствуют основным рискам проекта и уделяют различным его артефактам ровно такое внимание, которое требуется, исходя из их зрелости и важности для проекта.
  - Методы верификации, которые планируется применять, действительно способны дать лучшие результаты (с точки зрения обнаружения ошибок и получения достоверных оценок качества, отнесенных к затратам) в намеченных для них областях.
  - Подготовленные материалы (тесты, списки возможных ошибок для инспекций, формальные модели требований или окружения системы и пр.) соответствуют контексту использования системы, требованиям к проверяемым артефактам и связанным с ними проектным решениям и

могут быть использованы в качестве входных данных для выбранных методов проведения верификации.

## 2.4. Международные стандарты, касающиеся верификации ПО

Основным стандартом, регулирующим планирование и проведение верификации ПО, является стандарт IEEE 1012 на процессы верификации и валидации [12]. Этот стандарт содержит следующую информацию.

- Описание наборов отдельных задач верификации, соответствующих разным видам деятельности в рамках процессов, определенных в ISO 12207 [15]. Эти задачи выделены таким образом, чтобы как можно детальнее и полнее оценить результаты соответствующих видов деятельности. Для каждой задачи определены входные и выходные артефакты и рекомендуемые для использования в ее рамках техники верификации. Многие задачи выполняются многократно в рамках различных процессов жизненного цикла, для оценки разных артефактов. Отметим следующие задачи.
    - Подготовка планов проведения верификации и оценка их соответствия требованиям к ПО, ресурсам проекта и его рискам, а также используемым технологиям.
    - Оценка всех основных артефактов жизненного цикла — концепции системы, описания требований, проектных решений, кода и документации — в соответствии с описанными выше критериями.
    - Анализ критичности — определение уровня критичности отдельных требований, проектных решений, модулей, элементов кода и документации, связанных с ними рисков, их важности для достижений целей проекта, безопасности и экономической эффективности разрабатываемой или сопровождаемой системы.
    - Анализ привязки требований и их прослеживание — определение связей между требованиями, проектными решениями, отдельными модулями и функциями, тестами, разделами документации. Выполняется для проверки того, что все требования находят адекватное, соответствующее

их важности отражение во всех артефактах жизненного цикла, а все элементы этих артефактов построены для реализации требований.

  - Анализ интерфейсов — оценка корректности интерфейсов модулей и отдельных функций с точки зрения полноты, адекватности и точности отражения реализованных в них требований и проектных решений.
  - Анализ возможных сбоев — выделение наиболее рискованных и критичных ситуаций при работе ПО с точки зрения безопасности и экономической эффективности его эксплуатации, оценка возможного ущерба от них и их вероятности.
  - Анализ защищенности — анализ возможностей несанкционированного доступа к данным, коммуникациям и исполняемым модулям системы, возможностей получения контроля над ее работой в различных аспектах, а также оценка связанных с этим рисков.
  - Подготовка вспомогательных артефактов для верификации, прежде всего, разработка системных, интеграционных и модульных тестов.
  - Выполнение тестов всех уровней и анализ их результатов.

- Рекомендуемый шаблон плана проведения верификации и валидации, с описанием содержания основных его разделов. Этот план предполагает технически обоснованный выбор методов верификации в начале проекта, б***о***льшая часть его пунктов просто соответствует выделенным в стандарте задачам верификации.

  Для помощи при создании планов проведения верификации и валидации ранее был принят стандарт IEEE 1059 [33], но его содержание полностью покрывается IEEE 1012.

- Определение 4-х уровней критичности ПО (software integrity levels), от высокой (сбой в таких системах способен привести к невосполнимому ущербу — потери человеческой жизни, разрушению инфраструктуры, большим экономическим потерям) до минимальной (сбои имеют пренебрежимые последствия). Для каждого уровня определен минимальный набор задач верификации, которые рекомендуется выполнять для такого ПО, причем указываются также

рекомендуемая полнота проводимых проверок и уровень их формальности. Стандарт допускает также определение дополнительных уровней критичности.

Отдельные методы и задачи верификации и валидации описываются в следующих стандартах.

- IEEE 829 — документация тестирования ПО [34]. Это базовый стандарт, описывающий вспомогательные артефакты для тестирования. Основными такими артефактами являются следующие.
    - *Тестовый план* (test plan) — основной документ, связывающий разработку тестов и тестирование с задачами проекта. Определяет необходимые в проекте виды тестирования, используемые техники, проверяемые характеристики, компоненты системы, подлежащие тестированию, критерии оценки полноты тестов и критерии завершения тестирования на различных этапах, а также план выполнения отдельных действий по подготовке и проведению тестов и привязку ресурсов для этого. Тестовый план проекта должен корректироваться в соответствии с текущими задачами и рисками проекта, а также с появляющимися данными о качестве отельных составляющих системы.
    - *Тестовые варианты* (test case) — сценарии проведения отдельных тестов. Каждый тестовый вариант предназначен для проверки определенных свойств некоторых компонентов системы в определенной конфигурации и включает
        - действия по инициализации тестируемой системы (или компонента);
        - приведение ее в необходимое состояние, вместе с предыдущим шагом этот составляет *преамбулу* тестового варианта;
        - набор воздействий на систему, выполнение которых должно быть проверено данным тестом;
        - действия по проверке корректности поведения системы — сравнение полученных и ожидаемых результатов, проверка необходимых свойств результатов, проверка инвариантов данных системы и пр.;

    - финализацию системы — освобождение захваченных при выполнении теста ресурсов.

  Приведенное здесь описание тестового варианта несколько расширено по сравнению со стандартом IEEE 829.

  - *Описания тестовых процедур* (test procedure specifications). Тестовые процедуры могут быть представлены в виде скриптов или программ, автоматизирующих запуск тестовых вариантов, или в виде инструкций для человека, следуя которым можно выполнить те же варианты.
  - *Отчеты о нарушениях* (test incident reports) — детальное описание отдельных ошибок, обнаруженных при выполнении тестов, с указанием всех условий, необходимых для их проявления, нарушаемых требований и ограничений, возможных последствий, а также предварительной локализации ошибки в одном или нескольких модулях.
  - *Итоговый отчет о тестировании* (test summary report) — отчет с суммарной информацией о результатах тестов, включающей достигнутое тестовое покрытие по используемым критериям и общую оценку качества компонентов системы, проверяемых тестами.

- Модульное тестирование ПО регулируется стандартом IEEE 1008 [35], не пересматривавшимся с 1987 года. Этот стандарт достаточно детально описывает процедуру подготовки модульных тестов, их выполнения и оценки результатов, состоящую из 8-ми следующих видов деятельности.
  - Планирование, включающее в себя определение используемых методов тестирования, критериев полноты тестов и критерия окончания тестирования, а также определение необходимых ресурсов.
  - Определение проверяемых требований и ограничений.
  - Уточнение и детализация планов.
  - Разработка набора тестовых вариантов.
  - Выполнение тестов.
  - Проверка достижения критерия окончания тестирования и оценка полноты тестов по выбранным критериям.
  - Оценка затрат ресурсов и качества протестированных модулей.

- Группа стандартов ISO 14598 [36-41] описывают процессы оценки программных систем и компонентов, основанные на определении метрик и показателей, связанных с определенными характеристиками качества. В настоящий момент это действующие стандарты, но им на смену готовятся новые стандарты группы SQuaRE (см. ниже).
- Стандарт ISO 12119 [42] (и совпадающий с ним по содержанию IEEE 1465 [43]) описывает схему определения требований к различным артефактам жизненного цикла ПО и процесс их проверки с помощью тестирования. Он не действует с 2006 года в связи с пересмотром системы стандартов ISO, нацеленным на гармонизацию стандартов характеристик качества ПО ISO 9126 [21-24] и процессов их оценки ISO 14598 [36-41]. В результате должны появиться новые стандарты группы SQuaRE (Software Quality Requirements and Evaluation) [25]. В частности, ISO 12119 заменен на ISO 25051 [44], который лучше согласован с моделью качества ISO 9126 и IEEE 829.
- Группа стандартов ISO 15504 [30,45-48] описывает метод SPICE (Software Process Improvement and Capability dEtermination) оценки и совершенствования процессов разработки программного обеспечения. Этот метод [49] основан на схеме CMMI [50,51] анализа возможностей организации по разработке качественного ПО на основе оценки используемых в ней процессов разработки. Планируется выпустить 9 частей этого стандарта, на настоящий момент (май 2008 года) выпущено только 5.
- Стандарт IEEE 1028 [52] описывает один из методов проведения верификации — *экпертизы* (review). В нем выделены основные виды экспертиз — организационные экспертизы, технические экспертизы, инспекции, сквозной контроль и аудиты, определены роли их участников и возможные методики выполнения (подробнее см. следующий раздел).

  В ходе экспертиз в качестве вспомогательных материалов часто используются списки возможных или наиболее важных видов ошибок в анализируемом ПО. Пример подобной классификации и методика ее построения зафиксированы в стандарте IEEE 1044 [53,54].

В целом сложившаяся на данный момент система международных стандартов не содержит целостного подхода к верификации ПО. Б*о*льшая часть этих стандартов создана на основе опыта разработки сложных программных комплексов для авиакосмической и оборонной отраслей в 70-80-х годах XX века и ориентирована на процессный подход к программной инженерии. Этот подход предполагает сопоставление всех производимых действий с общими стандартами на разработку ПО, без аккуратного учета специфики предметной области и конкретного проекта, а также детальное планирование и документирование всех операций. В имеющихся стандартах отражены лишь некоторые методы верификации, лучше всего — экспертизы. В частности, достаточно плохо описаны методы тестирования, указывается лишь структура документации и общие процессы подготовки тестов, без систематического изложения содержательных техник разработки тестов и методов оценки полноты тестирования. Описание таких техник можно найти в национальных стандартах, см., например, британский стандарт на тестирование программных компонентов BS 7925-2 [55]. Кроме того, в имеющихся стандартах совсем не затронуты методы верификации, основанные на привлечении формального анализа свойств ПО, что может быть оправдано достаточно ограниченной пока областью их применения на практике.

Российские стандарты программной инженерии почти всегда являются переводами соответствующих стандартов ISO или IEEE, и поэтому имеют примерно то же содержание.

## *3. Методы верификации программного обеспечения*

В данном разделе рассматриваются методы верификации ПО, в основном нацеленные на оценку технических артефактов жизненного цикла. Такие методы в имеющейся литературе разделяются на следующие группы.

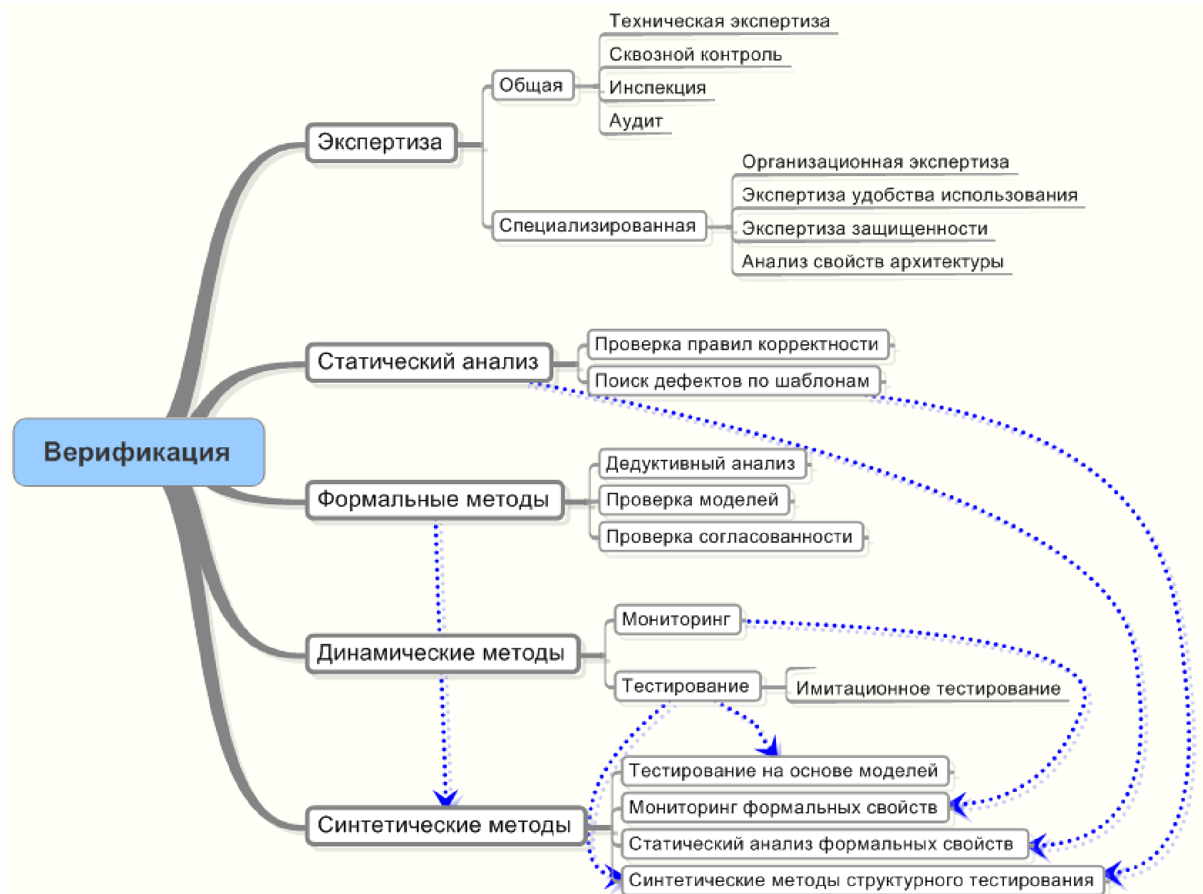


**Рисунок 4. Схема используемой классификации методов верификации.**

- *Экспертиза* (review) различных артефактов жизненного цикла ПО. Обычно в качестве видов экспертиз выделяют организационные экспертизы (management review), технические экспертизы (technical review), сквозной контроль (walkthrough), инспекции (inspection) и аудиты (audit). С середины 1990-х активно развиваются методы оценки архитектуры ПО на основе сценариев (scenario based software architecture evaluation), обычно не

соотносимые с «традиционными» экспертизами. От других методов верификации экспертизу отличает возможность выполнять ее, используя только сами артефакты жизненного цикла, а не их модели (как в формальных методах) или результаты работы (как в динамических).

Экспертиза применима к любым свойствам ПО и любым артефактам жизненного цикла и на любом этапе проекта, хотя для разных целей могут использоваться разные ее виды. Она позволяет выявлять практически любые виды ошибок, причем делать это на этапе подготовки соответствующего артефакта, тем самым минимизируя время существования дефекта и его последствия для качества производных артефактов. В то же время экспертиза не может быть автоматизирована и требует активного участия людей.

Эмпирические наблюдения показывают, что эффективность экспертиз в терминах отношения количества обнаруживаемых дефектов к затрачиваемым на это ресурсам несколько выше, чем для других методов верификации. Так, различные отчеты показывают, что от 50 до 90% всех зафиксированных в жизненном цикле ПО ошибок может быть обнаружено с помощью экспертиз [56-61]. За счет их раннего обнаружения может быть достигнута существенная экономия ресурсов — затраты на обнаружения ошибки составляют от 5 до 80% от таких же затрат при использовании тестирования [59-63]. Кроме того, регулярное участие в экспертизах является важным фактором в обучении сотрудников и способствует повышению качества результатов их работы.

В то же время эффективность экспертизы существенно зависит от опыта и мотивации ее участников [58-61], организации процесса, а также от обеспечения корректного взаимодействия между различными участниками. Это накладывает дополнительные ограничения на распределение ресурсов в проекте и может приводить к конфликтам между разработчиками, если руководство проекта обращает мало внимания на коммуникативные аспекты проведения экспертиз.

- *Статический анализ* свойств артефактов жизненного цикла ПО используется для проверки формализованных правил корректного построения этих артефактов и поиска часто встречающихся дефектов по некоторым шаблонам.

Такой анализ хорошо автоматизируется и может быть практически полностью возложен на инструменты, хотя иногда необходимо вручную определить, например, принятые в проекте стандарты кодирования. Однако применим он лишь к коду или к определенным форматам представления проектных артефактов, и способен обнаруживать только ограниченный набор типов ошибок.

Одной из известных проблем статического анализа является также следующая дилемма: либо используются строгие методы анализа, не допускающие пропуска ошибок (тех типов, что ищутся), но приводящие к большому количеству сообщений о возможных ошибках, которые таковыми не являются, либо точным является набор сообщений об ошибках, но возникает возможность пропустить ошибку.

Инструменты автоматической верификации на основе статического анализа применяются достаточно широко, поскольку не требуют специальной подготовки и достаточно удобны в использовании. Большинство техник статической проверки корректности программ, доказавших свою эффективность на практике, рано или поздно становятся частью компиляторов или даже преобразуются в семантические правила языков программирования.

- *Формальные методы верификации* используют для анализа свойств ПО формальные модели требований, поведения ПО и его окружения. Анализ формальных моделей выполняется с помощью специфических техник, таких как *дедуктивный анализ* (theorem proving), *проверка моделей* (model checking) или *абстрактная интерпретация* (abstract interpretation).

  Формальные методы применимы только к тем свойствам, которые выражены формально в рамках некоторой математической модели, а также к тем артефактам, для которых можно построить адекватную формальную модель. Соответственно, для использования таких методов в проекте необходимо затратить значительные усилия на построение формальных моделей. К тому же, построить такие модели и провести их анализ могут только специалисты по формальным методам, которых не так много, и чьи услуги стоят достаточно дорого. Построение формальных моделей нельзя автоматизировать, для этого

всегда необходим человек. Анализ их свойств в значительной мере может быть автоматизирован, и сейчас уже есть инструменты, способные анализировать формальные модели промышленного уровня сложности, однако чтобы эффективно пользоваться ими часто тоже требуется очень специфический набор навыков и знаний (в специфических разделах математической логики и алгебры).

Тем не менее, в ряде областей, где последствия ошибки в системе могут оказаться чрезвычайно дорогими, формальные методы верификации активно используются. Они способны обнаруживать сложные ошибки, практически не выявляемые с помощью экспертиз или тестирования. Кроме того, формализация требований и проектных решений возможна только при их глубоком понимании, и поэтому вынуждает провести тщательнейший анализ этих артефактов, для чего часто необходима совместная работа специалистов по формальным методам и экспертов в предметной области.
В последние 10 лет появились основанные на формальных методах инструменты [64-68], решающие достаточно ограниченные задачи верификации ПО из определенного класса, но зато способные эффективно работать в промышленных проектах и требующие для применения минимальных специальных навыков и знаний.

Гораздо чаще, чем к программам, формальные методы верификации на практике применяются к аппаратному обеспечению [69-72]. Их использование в этой области имеет более долгую историю, что привело к созданию более зрелых методик и инструментов. Это обусловлено более высокой стоимостью ошибок для аппаратного обеспечения, более однородной его структурой и более простыми примитивными элементами, более широким многократным использованием проектной информации, а также большей привычностью строгих ограничений и точных описаний для инженеров.

- *Динамические методы верификации*, в рамках которых анализ и оценка свойств программной системы делаются по результатам ее реальной работы или работы некоторых ее моделей и прототипов.

  Примерами такого рода методов являются обычное *тестирование* или

*имитационное тестирование*, *мониторинг*, *профилирование*.

Для применения динамических методов необходимо иметь работающую систему или хотя бы некоторые ее компоненты, или же их прототипы, поэтому нельзя использовать их на первых стадиях разработки. Зато с их помощью можно контролировать характеристики работы системы в ее реальном окружении, которые иногда невозможно аккуратно проанализировать с помощью других подходов. Динамически методы позволяют обнаруживать в ПО только ошибки, проявляющиеся при его работе, а, например, дефекты удобства сопровождения найти не помогут, однако, обнаруживаемые ими ошибки обычно считаются более серьезными.

Для применения динамических методов верификации обычно требуется дополнительная подготовка — создание тестов, разработка тестовой системы, позволяющей их выполнять или системы мониторинга, позволяющей контролировать определенные характеристики поведения проверяемой системы. Но системы тестирования, профилирования или мониторинга могут быть сделаны один раз и использоваться многократно для широких классов ПО, лишь сами тесты необходимо готовить заново для каждой проверяемой системы. В то же время подготовка тестов на ранних этапах создания ПО позволяет обнаружить множество дефектов в описании требований и проектных документах — фактически, разработчики тестов вынуждены в ходе своей деятельности выполнять экспертизу артефактов, служащих основой для тестов. Создание набора тестов, позволяющих получить адекватную оценку качества сложной системы, является довольно трудоемкой задачей. Однако среди разработчиков промышленного ПО сложилось (не вполне верное) мнение, что тестирование является наименее ресурсоемким методом верификации, поэтому на практике оно используется для оценки свойств ПО очень широко. При этом чаще всего применяются не слишком надежные, но достаточно дешевые техники, такие как (нестрогое) вероятностное тестирование, при котором тестовые данные генерируются случайным образом, или же тестирование на основе простейших сценариев использования, проверяющие лишь наиболее простые ситуации.

- *Синтетические методы.*

  В последние 10-15 лет появилось множество исследовательских работ и инструментов, в рамках которых применяются элементы нескольких перечисленных выше видов верификации. Так, в отдельные области выделились динамические методы, использующие элементы формальных, — *тестирование на основе моделей* (model-based testing, model driven testing) [73] и *мониторинг формальных свойств* (runtime verification, passive testing). Ряд инструментов построения тестов существенно использует как формализацию некоторых свойств ПО, так и статический анализ кода.

  Общая идея таких методов вполне понятна — попытаться сочетать преимущества основных подходов к верификации, купировав их недостатки.

Представленная здесь классификация методов верификации скорее обусловлена историческими причинами, чем основана на существенных характеристиках самих этих методов. Исследователи и разработчики новых методов и инструментов, пытаясь соотнести свои работы с работами предшественников, обычно ищут их в рамках одной из указанных групп, поэтому в настоящий момент такая схема достаточно удобна. Однако, как уже было сказано, в последнее время создаются синтетические методы, сочетающие элементы всех остальных, и через 5-10 лет, после появления достаточно большого количества таких подходов, потребуется более детальная и содержательная классификация методов верификации.

Далее при обзоре отдельных методов верификации методы, относящиеся к одному типу, рассматриваются вместе. Чаще всего при этом описывается более-менее детально только один представитель этого типа, а также указываются характеристики, по которым методы того же типа могут отличаться друг от друга.

## 3.1. Экспертиза

*Экспертизой* ПО (software review, переводится также как критический анализ, рецензирование, просмотр, обзор, оценка и просто анализ) называют все методы верификации, в которых оценка артефактов жизненного цикла ПО выполняется людьми, непосредственно анализирующими эти артефакты.

Традиционно выделяют следующие виды экспертиз.

- *Техническая экспертиза* (technical review) — систематический анализ артефактов проекта квалифицированными специалистами для оценки их внутренней согласованности, точности, полноты, соответствия стандартам и принятым в организации процессам, а также соответствия друг другу и общим задачам проекта.
- *Сквозной контроль* (walkthrough) — метод экспертизы, в рамках которого один из членов команды проверки представляет ее участникам последовательно все характеристики проверяемого артефакта, а они анализируют его, задавая вопросы, внося замечания, отмечая возможные ошибки, нарушения стандартов и другие дефекты.
- *Инспекция* (software inspection) — последовательное изучение характеристик артефакта, обычно следующее некоторому плану, с целью обнаружения в нем ошибок и дефектов.
- *Аудит* (audit) — анализ артефактов и процессов жизненного цикла, выполняемый людьми, не входящими в команду проекта, для оценки соответствия этих артефактов и процессов задачам проекта, заключенному контракту, общим стандартам, друг другу и пр.

Термины «экспертиза» (review) и «инспекция» (inspection) часто употребляются для описания всех перечисленных методов верификации, четкого различия между ними большинство работ не проводит. В рамках этого обзора, так же, как и в стандарте IEEE 1028 [52] инспекцией считается более формализованная и поверхностная проверка, нацеленная на обнаружение формальных дефектов (подобно техническому осмотру автомобиля) — обычно в ее ходе по некоторому контрольному списку проверяется формальное наличие определенных свойств и отсутствие определенных видов дефектов.

### 3.1.1. Оценка ПО по Фагану

Исторически первой техникой экспертиз стала оценка ПО по Фагану [74] (Fagan software inspection), использованная M. Fagan'ом в одном из проектов разработки ПО в IBM в 1972 году. Фактически, она представляет собой образец организации работ

(organizational pattern или process pattern), используемый для решения типовой задачи оценки соответствия друг другу двух артефактов жизненного цикла ПО. Эта техника определяет 4 роли участников и 6 шагов выполнения оценки. В ходе оценки сопоставляются первичный документ и вторичный документ, для создания которого первичный является входным. Список возможных комбинаций (не исчерпывающий) первичного и вторичного документов указан в таблице 1. Роли участников таковы.

- *Ведущий* (moderator). Он руководит подготовкой и проведением оценки, проведением собраний, назначает сроки выполнения работ, фиксирует обнаруженные дефекты, следит за готовностью входных данных для оценки и исправлением найденных ошибок после нее. В качестве ведущего должен использоваться компетентный разработчик или архитектор, не вовлеченный в проект, материалы которого оцениваются.
- *Автор* (author). Это автор первичного документа или человек, имеющий достаточно полное представление о нем. Его обязанности — подготовить рассказ об основных положениях первичного документа и отвечать на вопросы, возникающие у членов команды оценки по его поводу.
- *Интерпретатор* (reader). Это автор вторичного документа, который разработан в соответствии с первичным. Его обязанности — объяснить участникам инспекции основные идеи, лежащие в основе его интерпретации первичного документа, и отвечать на их вопросы по поводу вторичного документа.
- *Оценщик* (tester). В ходе всей оценки он анализирует вторичный документ, проверяя его на соответствие первичному и выявляя возможные несоответствия.

Процесс выполнения оценки состоит из следующих шагов.

**Таблица 1. Первичные и вторичные документы в рамках разных деятельностей.**

| Вид деятельности | Первичные документы | Вторичные документы |
|---|---|---|
| Анализ требований | Модели предметной области, концепция системы, составленные заказчиками и пользователями требования | Описание требований к ПО |
| Проектирование | Требования к ПО | Описание архитектуры, проектная документация |
| Кодирование | Проектная документация | Код, проектная документация на |

| | | отдельные компоненты |
|---|---|---|
| Тестирование | Требования к ПО, проектная документация, код | Тестовые планы и наборы тестовых вариантов |

1. *Планирование* (planning). На этом шаге ведущий должен убедиться в том, что первичный и вторичный документы готовы к проведению оценки — они существуют, написаны достаточно понятно, с нужной степенью детализации. Кроме того, на этом шаге проводится планирование всего хода оценки — определяются участники, их роли, назначаются сроки проведения собраний и время, выделяемое на выполнение каждого шага.
2. *Обзор* (review). Проводится собрание, на котором автор представляет наиболее существенные положения первичного документа и отвечает на вопросы участников о нем.

   Первичный и вторичный документы выдаются на руки участникам оценки для дальнейшей работы.

   Ведущий объясняет задачи данной оценки, вопросы и моменты, на которые стоит обратить особое внимание, а также сообщает, какие ошибки были уже обнаружены в рассматриваемых документах, чтобы участники группы имели представление об их проблемных местах.
3. *Подготовка* (preparation). Каждый из участников тщательно изучает оба документа самостоятельно, пытаясь понять заложенные в них решения и проследить их реализацию, фиксируя свои замечания к документам, неясные места, возможные дефекты.
4. *Совместная оценка* (inspection meeting). Проводится совместное собрание, на котором интерпретатор рассказывает об основных идеях и техниках, использованных во вторичном документе, а также объясняет, почему были приняты те или иные решения и как реализованы положения первичного документа.

   Участники задают вопросы и акцентируют внимание на проблемных местах. Если кто-то по ходу собрания замечает ошибку, ведущий должен убедиться, что все участники согласны считать ее ошибкой, т.е. несоответствием между

первичным и вторичным документами. Каждая ошибка фиксируется, описывается ее положение, она классифицируется по некоторой схеме, например, критическая (приводящая к ошибке в работе системы) или некритическая (связанная с опечатками, излишней сложностью или неудобством интерфейса и пр.).

5. *Доработка* (rework). В ходе доработки интерпретатор исправляет обнаруженные ошибки.
6. *Контроль результатов* (follow-up). Результаты доработки проверяются ведущим. Он проверяет, что все найденные ошибки были исправлены и что не было внесено новых ошибок. Если по результатам инспекции было переработано более 5-10% вторичного документа, следует провести полную инспекцию вновь, иначе ведущий сам определяет, насколько документ подготовлен к дальнейшему использованию.

   Кроме того, ведущий подготавливает отчет обо всех обнаруженных ошибках для последующего использования в других оценках и при контроле качества результатов разработки.

Оценка ПО по Фагану является разновидностью сквозного контроля (хотя по-английски она называется Fagan software inspection) — она более четко структурирована, чем техническая экспертиза, и выполняет систематическую проверку характеристик вторичного документа.

### 3.1.2. Другие виды общих экспертиз

Многочисленные техники проведения экспертиз [58-61], созданные после оценки ПО по Фагану, часто повторяют ее основные элементы. Отличия возникают по следующим параметрам.

- *Выделяемый набор ролей*. Часть техник определяет дополнительные роли к перечисленным выше, например, разделяют автора первичного документа и *докладчика*, представляющего его содержание. Иногда вводится *секретарь*, выполняющий вместо ведущего фиксацию обнаруженных дефектов.
- *Выделяемый набор шагов*. Планирование не считается во многих работах отдельным шагом. Обзорное собрание также часто не рассматривается как

обязательное [58,75,76], хотя иногда оно необходимо для более глубокого знакомства участников с целями проводимой проверки. Некоторые техники считают необходимым проведение еще одного собрания по окончании доработки для анализа причин возникновения дефектов.

- *Размер команды*. В целом, рекомендуется составлять небольшие команды, не более 6 человек. Отмечено, что увеличение количества участников более 5 приводит к большим затратам на обеспечение взаимодействия и адекватную передачу информации. Для оценки по Фагану оптимальной считается команда из 4 человек [75]. Часть техник рекомендует ограничиваться 2-мя людьми, некоторые рассчитаны только на индивидуальную работу.
- *Количество сессий проверки*. Иногда проводят несколько сессий анализа, в ходе каждой проверяя артефакт полностью снова. Такой подход позволяет выявить упущенные в ходе однократной проверки дефекты.
- *Необходимость проведения и количество общих собраний*. Фаган настаивал на необходимости общего собрания, указывая на его синергетический эффект при поиске дефектов. Многие авторы (например, [77]), однако, считают общие собрания необязательными, и, более того, повышающими затраты на проведение оценки без значимого эффекта для выявления дефектов. Отмечаемые коммуникативные проблемы при проведении собраний включают следующие: наличие «безбилетников», участников, не вносящих никакого вклада в обсуждение; следование за мнением большинства; боязнь высказать «глупое» мнение; перенесение внимания на одни вопросы в ущерб другим; доминирование одного из участников.
- *Техника работы с документом и использование вспомогательных материалов при анализе*. Большинство работ считают необходимым использование специальных техник чтения документов, повышающих эффективность их анализа. Часто используются контрольные списки возможных (наиболее важных) видов дефектов и списки аспектов корректности, подлежащих проверке, иногда они оформляются в виде наборов вопросов, на которые необходимо дать ответ (checklists). Отмечается, что вопросы (описания дефектов) должны быть сформулированы максимально точно, а весь список

должен размещаться на одной странице, чтобы быть обозримым для оценщика. В [78] предложена техника чтения кода, названная чтением с постепенным обобщением (reading by stepwise abstraction). При ее использовании для анализируемой последовательности инструкций кода строится функция, которую они вычисляют. При наращивании последовательности эта функция постепенно уточняется и трансформируется.

Еще одна возможная техника чтения — анализ по сценариям (scenario-based reading) [79], в ходе работы по которой оценщик следует заранее сформулированным рекомендациям по выявлению дефектов или пытается ответить на последовательность связанных вопросов.

- *Инструментальная поддержка*. Наиболее существенными факторами эффективности экспертиз являются опыт и мотивация участвующих в них людей [58-71], и чаще всего экспертиза проводится без использования инструментов, однако есть ряд программных средств, поддерживающих ее выполнение, решая вспомогательные задачи. Основные функции таких инструментов следующие.
    - Предоставление доступа к тексту документов (или диаграммам графических моделей) и возможности записывать замечания и вопросы, привязывая их к определенным элементам документа.
    - Предоставление удобного доступа к вопросникам и сценариям работы, обычно располагаемым на одном экране с анализируемыми документами.
    - Поддержка определения ролей участников, планирования работ и собраний для ведущего.
    - Поддержка обмена сообщениями и информацией о дефектах между участниками группы оценки.
    - Сбор и хранение информации о найденных дефектах.

Обзоры имеющихся инструментов поддержки экспертиз можно см. в [60,61].

В таблице 2 сопоставлены несколько методов проведения экспертиз.

**Таблица 2. Характеристики некоторых методов проведения экспертиз.**

| Метод | Размер команды | Сессии проверки | Техника чтения | Общие собрания | Пост-анализ |
|---|---|---|---|---|---|
| Fagan [74] | 3-5 | 1 | — | 1-2 | — |
| Gilb, Graham[58] | 4-6 | 1 | вопросники | 2-3 | Есть |
| Bisant, Lyle [80] | 2 | 1 | — | 1 | — |
| Оценка без собраний [77] | Индивиду-альная работа | 1 | — | Нет, есть встречи двух участников | — |
| Активная оценка проекта (active design review) [81] | 2 | > 1 одна сессия для каждой части артефакта | сценарии на основе вопросов | 1 | — |
| Britcher [82] | — | 4, параллельно | сценарии | 1-2 | — |
| Фазированные инспекции [83] | 1-2 | > 1, последовательно | вопросники | 1 | — |
| N-кратная оценка [63,84] | 3 в каждой команде, несколько команд | > 1, параллельно | — | 1 | — |

### 3.1.3. Специализированные методы экспертиз

Рассматривавшиеся до сих пор методы экспертиз являются общими — они применимы практически к любым артефактам жизненного цикла ПО. Однако, помимо таких методов, разработано довольно значительное число специализированных методов экспертиз. Одним из традиционных таких методов является организационная экспертиза.

*Организационная экспертиза* (management review) — это систематический анализ процессов жизненного цикла и видов деятельности, а также организационных артефактов проекта, выполняемый руководством проекта или от его имени для контроля текущего состояния проекта и оценки выполняемой в его рамках организационной деятельности на соответствие основным целям проекта [52].

Другим примером специализированного метода является *эвристическая оценка (инспектирование)* пользовательского интерфейса [85,86], нацеленная на оценку

удобства использования ПО. Она организуется как систематическая оценка различных элементов и аспектов интерфейса с точки зрения определенных эвристик. В качестве таких эвристик можно использовать определенный набор правил и принципов построения удобных в использовании интерфейсов [86], или любую достаточно полную систему эвристик, приводимых в руководствах по удобству использования ПО.

В рамках одного сеанса оценка интерфейса проводится несколькими специалистами, имеющими опыт в деятельности такого рода. Число оценщиков —3-5. Их результаты объединяются в общую картину. В процессе инспектирования разработчики должны отвечать на вопросы, касающиеся различных аспектов как предметной области, так и работы проверяемой программы.

Оценка проводится в два этапа. На первом исследуется архитектура интерфейса в целом, на втором — отдельные контексты взаимодействия и элементы интерфейса. В целом оценка занимает 1-3 часа, после чего проводится анализ полученных результатов, во время которого оценщики описывают обнаруженные ими проблемы и предлагают способы их устранения.

При других видах оценки удобства использования могут использоваться различные роли в группе оценщиков (оценщики, ведущий, секретарь, пользователи, разработчики, эксперты в предметной области), различные шкалы серьезности обнаруженных дефектов (не более 3-4-х уровней), метрики для оценки целостности, согласованности, адекватности интерфейса решаемым задачам [86].

Специализированным видом экспертизы является также экспертиза или аудит защищенности программных систем. При проведении экспертиз защищенности используются базы данных известных уязвимостей и дефектов защиты [87,88], которые могут быть специфичны для области применения проверяемого ПО и его типа (сетевое приложение, приложение на основе базы данных, компилятор, операционная система и пр.). Важную роль играют также возможные сценарии атак с учетом их вероятностей, стоимости для взломщиков и возможного ущерба для системы.

Более детальную информацию о методах проведения экспертизы защищенности ПО можно найти в [89,90]. Необходимо отметить, что защищенность информации обеспечивается не только ПО, но и административными процедурами в организациях,

где оно используется, и использующими ее людьми, и систематический анализ защищенности системы должен включать анализ и этих аспектов.

### 3.1.4. Методы анализа архитектуры ПО

Особое место среди специализированных методов экспертиз занимают систематические методы анализа архитектуры ПО [91-93].

Первым таким методом, разработанным в 1993 году, стал SAAM (Software Architecture Analysis Method) [94,95]. Оценка или сравнение архитектур по этому методу выполняется следующим образом.

1. Определить набор сценариев взаимодействия пользователей или внешних систем с анализируемой. Каждый такой сценарий может использовать возможности, которые уже есть в системе, планируются для реализации или являются новыми. Сценарии должны быть значимы для конкретных заинтересованных лиц — пользователей, разработчиков, представителей заказчика или контролирующей организации, инженеров по сопровождению, и пр. Чем полнее набор сценариев, тем выше будет качество анализа.
2. Определить архитектуру (или несколько сравниваемых архитектур). Это должно быть сделано в форме, понятной всем участникам оценки. Обычно для этого применяются общеупотребительные графические языки, например, UML, однако можно использовать специализированные нотации.
3. Классифицировать сценарии. Для каждого сценария из набора должно быть определено, поддерживается ли он уже данной архитектурой (-ами) или для его поддержки нужно вносить в нее (них) изменения. Сценарий может поддерживаться, т.е. его выполнение не потребует внесения изменений ни в один из компонентов, или же не поддерживаться, если его выполнение требует изменений в описании поведения одного или нескольких компонентов или изменений в их интерфейсах. Поддержка сценария означает, что лицо, заинтересованное в его выполнении, оценивает степень поддержки как достаточную, а необходимые при этом действия — как достаточно удобные.
4. Оценить сценарии. Определить, какие из сценариев полностью поддерживаются рассматриваемыми архитектурами. Для каждого неподдерживаемого сценария

надо определить необходимые изменения в архитектуре — внесение новых компонентов, изменения в существующих, изменения связей и способов взаимодействия. Если есть возможность, стоит оценить трудоемкость внесения таких изменений.

5. Выявить взаимодействие сценариев. Определить, какие компоненты требуется изменять для неподдерживаемых сценариев; если требуется изменять один компонент для поддержки нескольких сценариев, эти сценарии называют взаимодействующими. Нужно оценить смысловые связи между взаимодействующими сценариями.

   Малая связанность по смыслу между взаимодействующими сценариями означает, что компоненты, в которых они взаимодействуют, выполняют слабо связанные между собой задачи и их стоит декомпозировать.

   Компоненты, в которых взаимодействуют много (> 2-х) сценариев, также являются возможными проблемными местами.

6. Оценить архитектуру в целом (или сравнить несколько заданных архитектур). Для этого надо использовать оценки важности сценариев и степень их поддержки архитектурой. Чем больше сценариев поддерживается, чем меньше трудоемкость изменения архитектуры для поддержки остальных сценариев, тем лучше эта архитектура.

SAAM нацелен, прежде всего, на оценку модифицируемости архитектуры и ее соответствия требованиям, представленным в виде сценариев. Другие методы анализа архитектуры пытаются учитывать и другие характеристики качества ПО, а также предоставить четкие критерии полноты набора используемых сценариев, которых В SAAM не хватает. Наиболее зрелыми на сегодняшний день являются методы SAAM и ATAM (Architecture Tradeoff Analysis Method) [96,97], оба разработаны в Институте программной инженерии (SEI) университета Карнеги-Меллон. Они оба апробированы во многих проектах, в отличие от большинства других предложенных методов. Кроме того, ATAM позволяет оценивать практически любые атрибуты качества ПО за счет привлечения вспомогательных техник на этапе анализа сценариев.

Информация о ряде методов анализа архитектуры ПО представлена в таблице 3, подробнее см. обзоры [91-93]. Инструментальная поддержка таких методов пока

достаточна слаба. Описано лишь два-три инструмента, поддерживающих работу по методам SAAM и ATAM [101], которые используются в исследовательских целях.

**Таблица 3. Характеристики некоторых методов анализа архитектуры ПО.**

| **Метод** | **Техника оценки** | **Оцениваемые характеристики** | **Доп. результаты** | **Используемые описания архитектуры** | **Возможности многократного использования информации** |
|---|---|---|---|---|---|
| SAAM [94] | Сценарии | Модифицируемость, соответствие требованиям | Проблемные места | Статическое представление | — |
| ATAM [96] | Сценарии, метрики, спец. техники для каждого атрибута качества | Любые, для оценки каждой характеристики привлекаются эксперты | Точки увязки — элементы архитектуры, влияющие на несколько характеристик | Статические и динамические представления | Техники анализа отдельных атрибутов качества |
| SBAR [98] | Сценарии, мат. модели, симуляция | Любые | — | Детальные представления всех проектных решений | — |
| ALMA [99] | Сценарии | Модифицируемость | Последствия изменений, оценка затрат на поддержку | — | — |
| CBAM [100] | Сценарии, сравнительные оценки, спец. техники для каждого атрибута | Любые | Явные оценки выгод и стоимости проектных решений | Статические и динамические представления | Техники анализа отдельных атрибутов качества |

### 3.2. Статический анализ

Методы статического анализа артефактов жизненного цикла можно разделить на два вида: контроль того, что все формализованные правила корректности построения этих артефактов выполнены, и поиск типичных ошибок и дефектов в них на основе некоторых шаблонов. Часто инструменты статического анализа используют оба типа проверок.

Чаще всего используется статический анализ исходного кода, см., например, список инструментов [102], выполняющих такой анализ, или детальный обзор [103] трех из них, PolySpace Verifier [104], Coverity Prevent [105] и Klocwork K7 [106], реализующих наиболее сложные виды анализа. Проверенные на практике правила корректности кода или шаблоны типичных ошибок переносятся в среды разработки, такие как Eclipse или Microsoft Visual Studio, и постепенно становятся семантическими правилами языков программирования, их проверка возлагается уже на компиляторы этих языков. Поэтому статический анализ можно считать наиболее широко применяемым методом верификации.

Если в проекте используются языки описания архитектуры или графические языки проектирования, построенные с их помощью артефакты можно также проверять с помощью специализированных инструментов, например [107], которые также постепенно встраиваются в широко используемые среды моделирования, такие как Rational Rose.

Поэтому методы верификации при помощи статического анализа либо уже прошли апробацию на практике и используются в коммерческих инструментах и широко применяемых инструментах разработки общего назначения, либо все еще остаются в ранге новаторских, исследовательских работ. Исследовательские методы на данный момент, в основном, связаны с формализацией различных характеристик и свойств ПО и поэтому рассматриваются в разделе, посвященным синтетическим подходам к верификации.

## 3.3. Формальные методы верификации

Формальные методы верификации ПО используют формальные модели требований, поведения и окружения ПО для анализа его свойств. Такие модели являются либо *логико-алгебраическими*, либо *исполнимыми*, либо *промежуточными*, имеющими черты и логико-алгебраических, и исполнимых моделей. Прежде чем перейти к обсуждению методов анализа таких моделей, рассмотрим различные типы формальных моделей более подробно.

### 3.3.1. Логико-алгебраические модели

*Логико-алгебраические модели* (property-based models), они же — логические или алгебраические исчисления. При моделировании ПО модель такого типа описывает некоторый набор его свойств, быть может, изменяющийся со временем, но не дает точного представления о том, за счет чего изменяются эти свойства.

Отличие между логическими и алгебраическими исчислениями довольно условно, но, несколько упрощая, можно считать, что логика имеет дело с утверждениями в рамках какого-то языка, а алгебра — с равенствами и неравенствами построенных в некотором языке выражений. В первом случае основным объектом внимания являются утверждения, ложные или истинные, а во втором — выражения или термы, относящиеся к какому-то типу.

Примеры логических исчислений таковы.

- *Исчисление высказываний* (пропозициональное исчисление, propositional calculus) [108,109]. В нем есть атомарные высказывания, возможно, зависящие от объектных переменных, а также логические связки & («и», конъюнкция), $\vee$ («или», дизъюнкция), $\neg$ («не», отрицание), $\Rightarrow$ («следовательно», импликация), $\Leftrightarrow$ (эквивалентность), с помощью которых можно из одних высказываний строить другие, более сложные.
- *Исчисление предикатов* (predicate calculus) [108,109] добавляет в исчисление высказываний возможность использовать кванторы по объектным переменным для построения новых утверждений. Кванторы в этом исчислении бывают двух

видов — ∀ «для любого» и ∃ «существует».

В исчислении предикатов помимо объектных переменных есть функциональные и предикатные. Первые представляют собой разнообразные функции, результат применения функции к объекту является объектом. Вторые представляют неопределенные утверждения, результат их применения к объекту должен всегда быть истиной или ложью. В нетипизированном исчислении все объекты равноправны, функции и предикаты могут принимать любые объекты в качестве аргументов. В типизированных исчислениях каждый объект имеет тип, а функциональные и предикатные переменные — сигнатуру, т.е. список типов параметров и тип результата для функций. Соответственно, строить формулы можно только соблюдая соответствие типов параметров типам выражений, подставляемых на место этих параметров.

- *Исчисления предикатов более высоких порядков* (higher-order calculi). В этих исчислениях можно использовать кванторы не только по объектным переменным, но и по функциональным или предикатным.

  Например, определение равенства объектов иногда формулируется так: $x = y \equiv \forall P\ P(x) \Leftrightarrow P(y)$, два объекта равны, если любое утверждение одновременно выполнено или не выполнено для них обоих.

- *λ-исчисление* (lambda calculus) [110] — с помощью лямбда-оператора позволяет строить функции из выражений, например, выражение $\lambda x\ x*x$ обозначает функцию возведения в квадрат. В этом примере x также является связанной, не имеющей собственного содержания переменной.

  В λ-исчислении с типами, так же как и в типизированном исчислении предикатов, объекты имеют типы, а функции — сигнатуры.

  *λ-исчисления более высоких порядков* [111] позволяют применять λ-оператор не только к объектам, но и к типам. При этом получаются функции, преобразующие типы в типы. Примером такого преобразования является построение по типу T типа списка объектов типа T.

  Часто расширения исчисления предикатов и λ-исчисления первого порядка

совместно называют *логиками первого порядка*, а расширения исчисления предикатов и λ-исчисления высших порядков — *логиками высших порядков*.

- *Модальные логики* [112] помимо связок допускают построение утверждений с помощью операторов с дополнительной смысловой нагрузкой, давая возможность строго анализировать связи между, например, утверждениями «x = 3», «доказано, что x = 3», «может быть, что x = 3», «всегда будет x = 3» или «хотелось бы, чтобы было x = 3».
- Специальным случаем модальных логик являются *временные логики* (temporal logics) [111,113,114], в которых дополнительные операторы используются для описания временной последовательности событий — «как только x станет равно 3, y должно стать равно 0», «после того, как x станет больше 0, спустя некоторое время y обязательно станет равно 5».

  В логиках с дискретным временем считается, что моменты времени отделены друг от друга и у каждого момента есть следующий. Утверждение **X** P означает, что утверждение P будет выполнено в следующий момент времени, а **G** P — что P будет выполнено во все будущие моменты времени. Первый пример из предыдущего абзаца можно записать как **G** (x = 3 ⇒ y = 0), а второй — как **G** (x > 0 ⇒ ¬**G** ¬(y = 5)), т.е. «после того, как x станет положительным, y не может все время оставаться не равным 5». Конструкция ¬**G** (¬P) имеет смысл «когда-нибудь в будущем P выполнится» и для нее есть отдельная запись **F** P.
- Дальнейшим обобщением *логики линейного времени* (linear temporal logic, LTL), примеры формул которой приведены в предыдущем пункте, является *логика дерева вычислений* (Computation Tree Logic, CTL*) [111,113,114], в которой можно делать утверждения не только об отдельных событиях и их порядке во времени, но и о различных возможностях развития событий.
- Еще более общим исчислением, чем временные логики, является *μ-исчисление* (или исчисление неподвижных точек) [111,113,114].
- Временные логики используют время неявно, только для рассмотрения возможных последовательностей событий. Если необходимо явно учитывать величину интервалов времени между событиями, используются *логики явного*

*времени* (timed temporal logics), в утверждениях которых участвуют переменные, обозначающие некоторые моменты времени, и явно указанные величины временных интервалов.

В качестве примеров алгебраических моделей приведем следующие.

- *Реляционные алгебры* [115,116], лежащие в основе реляционных систем управления базами данных.
- *Алгебраические модели абстрактных типов данных*. Например, можно определить абстрактный тип данных Stack стека объектов типа T как алгебру с набором операций pop: Stack → Stack×T и push: Stack×T → Stack, удовлетворяющую соотношению pop(push(s, t)) = (s, t). В принципе, все свойства стека можно вывести из этого соотношения. Аналогичное описание для очереди объектов типа T, несмотря на небольшое содержательное отличие от стека, выглядит значительно сложнее.
- *Алгебры процессов* (*исчисления процессов*, process calculi) [117-123]. Это алгебраические исчисления, объектами которых являются события и процессы, создающие события или реагирующие на них. Обычно для процессов определены операции последовательной (‘;’) и параллельной (‘||’) композиции и операция выбора из двух альтернатив (альтернативная композиция, ‘+’). Процесс является моделью выполняющейся программы. Последовательная композиция процессов моделирует выполнение сначала первого из них, затем второго. Параллельная композиция моделирует параллельное выполнение обоих процессов. Выбор из двух процессов моделирует выполнение либо первого, либо второго. В большинстве таких исчислений процессы могут взаимодействовать, обмениваясь событиями (один процесс порождает событие, другой или другие его потребляют).

  Наиболее широко известны исчисления процессов CSP (Communicating Sequential Processes) [118], CCS (Calculus of Communicating Systems) [119], ACP (Algebra of Communicating Processes) [120,121]. Для моделирования мобильных вычислений, т.е. вычислений в среде с изменяющимися связями между узлами и с переносом самих вычислительных процессов с одного узла на другой, используют *π-исчисление* [122] и *исчисление окружений* (ambient calculus) [123].

### 3.3.2. Исполнимые модели

*Исполнимые модели* (или операционные, executable models) характеризуются тем, что их можно каким-то образом выполнить, чтобы проследить изменение свойств моделируемого ПО. Каждая исполнимая модель является, по сути, программой для некоторой достаточно строго определенной виртуальной машины.

Зачем моделировать свойства одних программ через свойства других? Для этого есть, по крайней мере, три причины.

- В модели обычно учитываются не все свойства моделируемого ПО, а только важные для рассматриваемой в данный момент задачи. Поэтому модели оказываются значительно проще моделируемых систем, их гораздо удобнее анализировать.
- Виртуальные машины используемых на практике языков программирования очень сложны и определены нечетко, а виртуальные машины моделей значительно более просты и обозримы. Это позволяет провести исчерпывающий анализ возможного поведения модели, выявить все классы возможных при ее работе ситуаций.
- За счет иного, «перпендикулярного» взгляда на систему часто можно увидеть такие ее характеристики и особенности, на которые ранее просто не обращали внимания.

Все виды исполнимых моделей можно считать расширением и обобщением *конечных автоматов*, поэтому их стоит рассмотреть в качестве первого примера.

- *Конечный автомат* (finite state machine, FSM) [124,125] представляет собой систему с конечным множеством состояний, среди которых выделено начальное состояние, и конечными множествами воспринимаемых извне стимулов (входных символов) и создаваемых реакций (выходных символов). Кроме этого, в конечном автомате определен некоторый набор переходов между состояниями, каждый переход помечен вызывающим его стимулом и выдаваемой реакцией.

  Пример конечного автомата с множеством состояний {0,1,2}, множеством стимулов {a,b} и множеством реакций {x,y} изображен на Рис. 5.

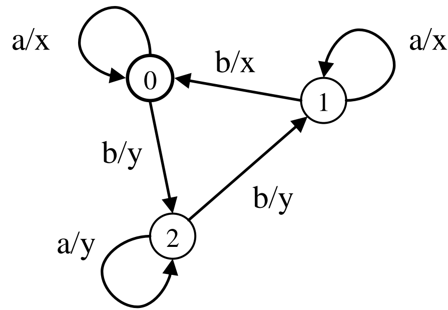


**Рисунок 5. Пример конечного автомата.**

Чтобы конечный автомат «заработал», ему нужно передать последовательность стимулов. Тогда автомат начинает считывать стимулы один за другим и выполнять переходы, помеченные этими стимулами, начиная с начального состояния. Выполнение перехода заключается в изменении текущего состояния и выдаче реакции, которой помечен данный переход. Так, если считать начальным состоянием показанного на Рис. 7 автомата состояние 0, то после подачи ему на вход последовательности abababab он перейдет в состояние 2 и выдаст последовательность реакций xyyyxxxy.

Можно считать, что каждый автомат реализует некоторое преобразование конечных последовательностей стимулов в конечные последовательности реакций. Помимо этого преобразования автомат задает и способ его реализации — одно и то же преобразование может быть реализовано разными конечными автоматами.

- Одним из обобщений конечных автоматов являются *конечные системы помеченных переходов* (или просто системы переходов, labeled transition systems, LTS) [126]. В системе переходов каждый переход помечен только одним символом — либо стимулом, либо реакцией (строго говоря, это верно для систем переходов со стимулами и реакциями, IOLTS, в случае общих LTS считается, что есть только один вид символов), либо может вообще не иметь метки (выполнение такого перехода никак не проявляется вовне).

  Любой автомат можно представить как систему переходов, на Рис. 6 слева показана система переходов, эквивалентная автомату, представленному на Рис. 5. Но системы переходов могут реализовывать более сложные

преобразования последовательностей символов, и поэтому не всякая система переходов представима как автомат — пример представлен на Рис. 6 справа.

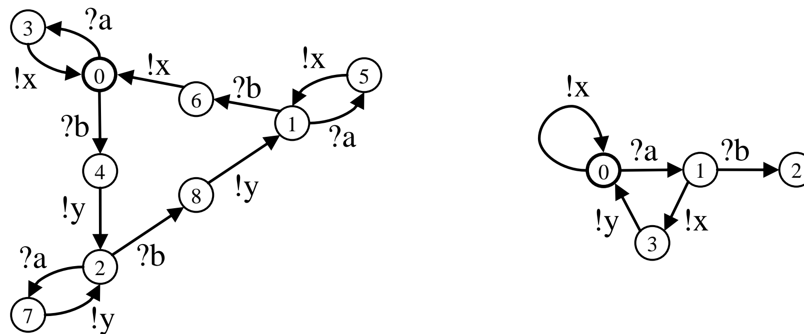


**Рисунок 6. Примеры конечных систем переходов.**

Системы переходов являются исполнимым аналогом алгебр процессов, для многих таких алгебр за счет определения подходящих видов стимулов и реакций можно каждый процесс представить как систему переходов.

- Другое обобщение конечных автоматов — *расширенные конечные автоматы* (extended finite state machines, EFSM) [127]. В расширенном автомате, помимо состояний (которые здесь называются состояниями управления) и переходов есть конечное множество внутренних переменных, способных принимать различные значения. Полное состояние расширенного автомата, обычно называемое его конфигурацией, включает текущее состояние управления и текущие значения всех переменных. Стимулы и реакции могут иметь параметры. Помимо стимула и реакции, каждый переход помечен также охранным условием (условием блокировки, предохранителем, guard condition) и действием (action).

  Охранное условие (указывается в квадратных скобках) зависит от переменных и параметров стимула и определяет дополнительное условие выполнения данного перехода. Т.е. для того, чтобы переход выполнился, мало подать на вход автомату нужный стимул, нужно еще использовать такие значения его параметров, чтобы охранное условие выполнилось.

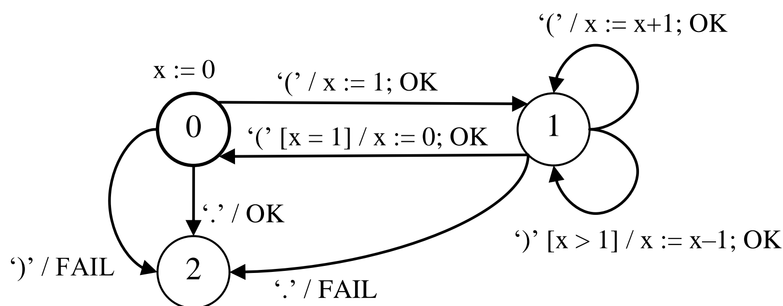


**Рисунок 7. Пример расширенного конечного автомата.**

Действие (указывается перед реакцией) определяет новые значения внутренних переменных и значения параметров реакции в зависимости от прежних значений переменных и значений параметров стимула.

На Рис. 7 показан пример расширенного автомата, который распознает правильно построенные скобочные конструкции, заканчивающихся точкой. Пока скобки расставлены правильно, этот автомат выдает OK, как только возникает неправильность, выдается FAIL.

- *Взаимодействующие автоматы* (communicating finite state machines, CFSM) представляют собой набор конечных автоматов, некоторые из которых связаны каналами для передачи реакций одного автомата как стимулов для другого.
- *Иерархические автоматы* (hierarchical state machines) позволяют определять переходы из группы состояний (но конечное состояние такого перехода должно быть только одно!), таким образом экономя на описании одинаковых переходов. Кроме того, в иерархических автоматах могут быть группы параллельных состояний — несколько групп состояний, объединяемых в параллельное семейство. Конечным состоянием некоторого перехода может быть такое семейство. Это означает, что после выполнения такого перехода автомат оказывается сразу в нескольких состояниях, по одному из каждой параллельной группы, входящей в это семейство, и может совершать несколько переходов параллельно, если в рамках этих групп есть переходы по одинаковым стимулам.
- *Временные автоматы* (timed automata) [128,129] — обычно это расширенные автоматы, содержащие дополнительный набор переменных-таймеров, значения которых изменяются сами по себе с течением времени. Значения таймеров

можно использовать в условиях переходов, изменении переменных или значениях параметров реакций. Кроме этого, таймеры можно запускать в действиях, связанных с переходами. Запущенный таймер начинает отсчет времени с 0. Время может быть дискретным или непрерывным, что позволяет моделировать поведение разнообразных систем реального времени.

- Еще один пример обобщенных автоматов — *гибридные автоматы* (hybrid automata) [130,131], применяемые для моделирования систем, взаимодействующих с непрерывными физическими процессами. В этих автоматах часть переменных обычно имеет значения, изменение которых описывается системой дифференциальных уравнений.
- Примером моделей с возможностью явного описания параллелизма являются *сети Петри* (Petri nets) [132-134]. В сети Петри есть состояния и переходы, но один переход может связывать два произвольных множества состояний (одно из них может быть пустым), соответственно, есть множество начал перехода и множество его концов. При изображении сетей Петри состояния показывают кружками, а переходы — линиями, «барьерами», которые могут соединяться входящими и выходящими стрелками только с состояниями. Кроме этого, есть набор меток или маркеров, которые расположены в состояниях сети и в ходе ее работы переходят из одних состояний в другие по переходам. Переход срабатывает, если в каждом состоянии, из которого он выходит, есть маркер, и при его срабатывании по одному маркеру из каждого входного состояния пропадает, а в каждом выходном состоянии добавляется по одному маркеру.

  Показанная на Рис. 8 сеть Петри моделирует работу блокирующего буфера, способного хранить два элемента. Переход с пустым множеством начал сверху представляет активность поставщика, который может попытаться положить что-то в буфер — при этом в состоянии p возникнет маркер. Аналогичный переход снизу представляет активность потребителя, который может попытаться забрать объект из буфера — тогда маркер возникает в состоянии c. Состояния 0, 1, 2 вместе с передвигающимся по ним маркером представляют состояние самого буфера, номер состояния с маркером совпадает с количеством объектов, лежащих в буфере.

Обобщения сетей Петри могут использовать несколько типов маркеров, дополнительную разметку переходов и состояний, кратные переходы и пр.

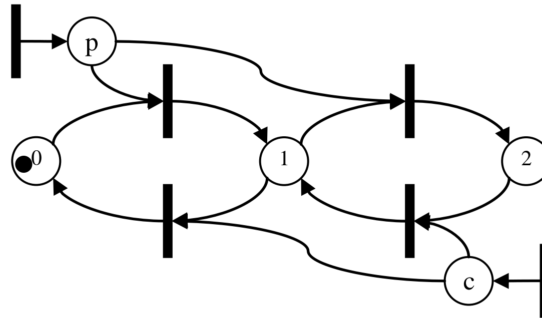


**Рисунок 8. Пример сети Петри.**

- Обычный конечный автомат можно рассматривать и как реализующий преобразование бесконечных последовательностей стимулов, тогда его принято называть *ω-автоматом* [135]. Обобщенные автоматы также могут иметь бесконечные множества состояний, стимулов или реакций, могут использовать несколько типов стимулов или реакций и т.п.

  Кроме того, все перечисленные выше дополнительные структуры могут использоваться в различных комбинациях, давая, например, взаимодействующие бесконечные временные системы переходов или расширенные иерархические сети Петри. Так диаграммы состояний в UML (Statecharts) [136] представляют собой взаимодействующие расширенные иерархические автоматы.
- Примером исполнимых моделей являются также *машины абстрактных состояний* (abstract state machines), введенные Ю. Гуревичем [137,138]. Каждая такая машина основана на сигнатуре некоторой универсальной алгебры, т.е. на наборе функциональных символов с определенным количеством параметров каждый. Состояния такой машины являются алгебрами с этой сигнатурой — множествами, в которых каждому функциональному символу сопоставлена некоторая операция с таким же, как у этого символа, числом параметров. Все возможные переходы описаны одной общей программой, состоящей из инструкций переопределения значения одного из функциональных символов на

некотором наборе аргументов, условных инструкций и инструкций параллельного выполнения одного и того же набора действий для всех объектов, удовлетворяющих некоторому условию.

### 3.3.3. Модели промежуточного типа

Модели промежуточного типа имеют черты как логико-алгебраических, так и исполнимых. Стоит отметить, что часть перечисленных выше примеров может быть по ряду причин отнесена к обоим этим классам, например, процессные алгебры имеют точный исполнимый аналог — системы помеченных переходов, а машины абстрактных состояний определяются как некоторое семейство универсальных алгебр.

Есть, однако, виды моделей, в которых логико-алгебраические и исполнимые элементы сочетаются более прямо. Основные их виды следующие.

- *Логики Хоара* (Hoare logics) [139] являются специфическим видом логик, утверждения которых состоят из формул логики некоторого вида и программных инструкций. В простейшем виде это тройки {Φ}P{Ψ}, где P — часть программы на определенном языке, а Φ и Ψ — формулы исчисления высказываний, зависящие от переменных, входящих в P. Φ интерпретируется как условие, выполнение перед началом исполнения P (предусловие), а Ψ — как условие, которое должно быть выполнено после этого исполнения (постусловие). Если Ψ действительно всегда истинно после исполнения P в состоянии, где истинно Φ, такая тройка тоже считается истинной.

  В логике Хоара для некоторого языка программирования семантика этого языка задается в виде правил вывода, которые позволяют выводить шаг за шагом истинные тройки Хоара из тавтологий, тождественно истинных высказываний.
- Обобщением логик Хоара являются *динамические* или *программные логики* (dynamic logics, program logics) [140,141]. Они являются специальным типом модальных логик, в которых операторы модальности связаны с инструкциями программ. Обычно используются операторы [P] и <P>, где P — некоторая программа. Утверждение [P]Φ означает, что всегда после выполнения программы P формула Φ истинна, а <P>Φ — что после выполнения P Φ может

оказаться истинной. Тройка Хоара $\{\Phi\}P\{\Psi\}$ может быть представлена в динамической логике как $\Phi \Rightarrow [P]\Psi$.

- *Программные контракты* (software contracts) [142,143], наоборот, являются частным случаем логики Хоара, сужающим возможности использования логических формул.

  Программный контракт представляет собой описание поведения набора программных компонентов представленное в виде описания сигнатур операций каждого из этих компонентов, структур их состояний, а также предусловий и постусловий для каждой операции и наборов инвариантов для каждого компонента в отдельности.

  Инвариант компонента является предикатом, зависящим от элементов состояния этого компонента, который должен быть выполнен в состояниях, когда ни одна из операций компонента не выполняется (т.е. либо еще не была вызвана, либо уже завершила работу). Инварианты описывают ограничения целостности внутренних данных компонента, и обязанность соблюдать их ложится на его реализацию.

  Предусловие операции компонента представляет собой предикат, зависящий от элементов состояния этого компонента и параметров этой операции, который должен быть выполнен при корректных обращениях к этой операции извне компонента. При вызове операции с нарушением предусловия ее поведение не определено. Предусловие является частью контракта, которую обязано соблюдать окружение компонента, чтобы обеспечить его корректную работу. Постусловие операции представляет собой предикат, зависящий от параметров операции, ее результата, элементов состояния компонента до вызова операции и тех же элементов после окончания ее работы. Постусловие должно выполняться сразу после окончания работы операции, вызванной с выполненным предусловием, и формализует ответственность реализации компонента за корректность его работы.

  Контракты часто невозможно выполнить непосредственно, поскольку постусловия не определяют прямо корректные результаты операций и следующие состояния, а лишь оценивают переданные им данные.

### 3.3.4. Классификация формальных методов

Для того, чтобы проверить выполнение тех или иных свойств с помощью формальных методов, необходимо формализовать свойства и проверяемый артефакт, т.е. построить формальные модели для того и другого. Модель проверяемых свойств принято называть *спецификацией*, а модель проверяемого артефакта — *реализацией*. Заметим, что здесь спецификация и реализация — термины, обозначающие формальные модели, а не описание требований и реализующий их набор программ, как обычно. После этого нужно проверить некоторое формально определенное соответствие или отношение этих моделей, которое моделирует выполнение данных свойств для данного артефакта.

Поскольку и реализация, и спецификация могут быть моделями двух основных видов, логико-алгебраическими или исполнимыми, возможно 4 разных комбинации этих видов моделей. Однако на практике никогда не используют для спецификации исполнимую модель в сочетании с логико-алгебраической для реализации, поскольку в такой комбинации невозможно обеспечить большую абстрактность спецификации по сравнению с реализацией. Таким образом, остаются три разных случая.

- И спецификация S, и реализация I представлены как логико-алгебраические модели. В этом случае выполнение специфицированных свойств в реализации моделируется отношением *выводимости*, что обычно записывается как $I \vdash S$.

  Чаще всего для его проверки используется метод *дедуктивного анализа* (theorem proving), т.е. проверки того, что набор утверждений, представляющий спецификацию, формально выводится из реализации и, быть может, каких-то гипотез о поведении окружения системы, сформулированных в том же формализме, что и реализация.
- Спецификация S является логико-алгебраической моделью, а реализация I — исполнимой. Выполнение специфицированных свойств в реализации в этой ситуации называется отношением *выполнимости* и записывается как $I \vDash S$.

  Для его проверки используется метод *проверки моделей* (model checking), в рамках которого чаще всего выполнимость проверяется непосредственным

исследованием всей реализации, или такой ее части, свойства которой полностью определяют свойства всей реализации в целом. Обычно эту работу выполняет не человек, а специализированный инструмент.

- И спецификация S, и реализация I представлены как исполнимые модели. В этом случае общепринятого названия или обозначения для выполнения специфицированных свойств в реализации нет — используются термины «симуляция» или «моделирование» (simulation), «сводимость» (reduction), «соответствие» или «согласованность» (conformance). Далее в этой работе используется последний термин — *согласованность*.

Для методов проверки согласованности тоже пока нет общепринятого названия.

В тех случаях, когда используются модели промежуточного типа, применяемый метод определяется теми составляющими модели, которые для него наиболее существенны. Так, логики Хоара и динамические логики чаще всего используют для дедуктивного анализа, а программные контракты могут применяться в различных методах.

### 3.3.5. Методы и инструменты дедуктивного анализа

Дедуктивный анализ используется для верификации логико-алгебраических моделей. Первые методы дедуктивного анализа программ были предложены Флойдом [144] и Хоаром [139] в конце 1960-х годов. В основе этих методов лежит логика Хоара и предложенная Флойдом техника доказательства завершения циклов, основанная на инвариантах цикла и монотонно изменяющихся в ходе его выполнения оценочных функциях. Выполнение верификации программы в методе Флойда организовано следующим образом.

1. Спецификация программы в виде ее предусловия и постусловия определяется формально, например, в рамках исчисления высказываний.
2. В коде программы или на ее блок-схеме выбираются точки сечения, так чтобы любой цикл содержал по крайней мере одну такую точку. Начало и конец программы (все возможные точки выхода из программы можно свести к одной) тоже считаются точками сечения.

3. Для каждой точки сечения i находится предикат $\Phi_i$, характеризующий отношения между переменными программы в этой точке. В начале программы в качестве такого предиката выбирается предусловие, в конце — постусловие. Кроме того, выбирается оценочная функция φ, отображающая значения переменных программы в некоторое упорядоченное множество без бесконечных убывающих цепей (например, натуральные числа).
4. В результате программа разбивается на набор возможных линейных путей между парами точек сечения. Для каждого такого пути $P_{ij}$ между точками i и j нужно проверить истинность тройки $\{\Phi_i\}P_{ij}\{\Phi_j\}$. Если это удается, программа частично корректна, т.е. работает правильно, если завершается.
5. Для каждого простого цикла (начинающегося и заканчивающегося в одной и той же точке), нужно найти на нем такой путь $P_{ij}$, для которого можно доказать $\{\Phi_i \;\&\; \varphi = a\}P_{ij}\{\Phi_j \;\&\; \varphi < a\}$ для некоторой дополнительной переменной a. Это позволяет утверждать, что цикл завершится, поскольку значения оценочной функции не могут уменьшаться неограниченно.

Достаточно трудной задачей в методе Флойда является нахождение подходящих предикатов. Для ее упрощения Дейкстра [145] предложил использовать технику построения слабейших предусловий, т.е. для программы P и формулы Ψ строить такую формулу wp(P,Ψ), что выполнено {wp(P,Ψ)}P{Ψ} и для любой формулы Φ, если {Φ}P{Ψ}, то $\Phi \Rightarrow wp(P,\Psi)$. Для верификации программы при этом достаточно показать, что ее слабейшее предусловие при заданном постусловии следует из исходного предусловия.

В таком виде метод Флойда применим лишь к довольно ограниченному классу программ — в них не должно быть массивов или указателей, вызова подпрограмм, параллелизма, взаимодействия с окружением по ходу работы. В дальнейшем были предложены аналогичные методы для дедуктивной верификации более сложных программ, в том числе параллельных [146], с использованием более сложных моделей, например, процессных алгебр, вместо исчисления высказываний.

Дедуктивный анализ в принципе может быть выполнен человеком, но для практически значимых систем сам размер спецификации и реализации таков, что

необходимо использование специализированных инструментов для автоматического построения доказательств (provers) или предоставляющих существенную помощь в их осуществлении (proof assistants) [147]. Поэтому чаще всего в качестве формализма для представления проверяемых свойств и реализации выбирают формализм одного из этих инструментов. При этом часто приходится учитывать, что чем более выразительно исчисление, чем удобнее с его помощью формулировать необходимые утверждения, тем более сложной задачей является проверка их доказуемости.

Инструментов построения доказательств достаточно много, см., например, их список в Wikipedia [148] Они относятся к одному из следующих типов.

- Инструменты, основанные на расширениях пропозициональной логики или логик первого порядка.

  Для пропозициональной логики существуют алгоритмы, проверяющие правильность утверждений, но это NP-полная задача. Для исчисления первого порядка множество доказуемых утверждений неразрешимо, но перечислимо — можно с помощью алгоритма постепенно построить все доказуемые утверждения, но нельзя проверить, что утверждение недоказуемо.

  Чтобы получить достаточную для дедуктивного анализа программ выразительность часто используют расширения логики первого порядка различными простыми теориями, а также индуктивными правилами вывода, как явными, так и неявными, представленными в виде правил переписывания термов (term rewriting rules).

  В этой категории наиболее известны инструменты ACL2 [149,150] (инструмент Бойера-Мура, ранняя его версия называлась Nqthm), E [151,152] (и его ответвление E-SETHEO [153]), свободно доступный KeY [154,155], Vampire [156], Waldmeister [157], Darwin [158].

  Первые 4 инструмента в этом списке активно используются для формальной верификации ПО и аппаратного обеспечения, даже достаточно сложных систем. Последние три являются признанными лидерами состязаний инструментов автоматического доказательства теорем [159], но их практическое использование для верификации не столь широко [160].

- Инструменты, основанные на логиках высших порядков.

  Они всегда интерактивны, поскольку теоремы в логиках высших порядков даже не перечислимы. Во время проведения доказательства с их помощью часто требуется вмешательство человека, чтобы сформулировать нужную вспомогательную лемму или изменить стратегию доказательства. Эти инструменты могут отличаться друг от друга наличием и объемом вспомогательных теорий, доступных для использования.

  Наиболее известные и широко используемые из них — PVS [161,162], HOL [163,164] (является развитием LCF), Isabelle [165,166], Coq [167,168]. Два первых инструмента многократно применялись при верификации как программных, так и аппаратных систем, см., например, [69-71,169].

### 3.3.6. Методы и инструменты проверки моделей

Проверка моделей (model checking) [114] используется для проверки выполнения набора свойств, записанных в виде утверждений какого-либо логико-алгебраического исчисления на исполнимой модели, моделирующей определенные проектные решения или код ПО.

Чаще всего для описания проверяемых свойств используется некоторая временная логика или μ-исчисление, а в качестве модели, свойства которой проверяются, выступает конечный автомат, состояния которого соответствуют наборам значений элементарных формул в проверяемых свойствах, обычно он называется *моделью Крипке*. Проверку модели выполняет специализированный инструмент, который либо подтверждает, что модель действительно обладает заданными свойствами, либо выдает сценарий ее работы, в конце которого эти свойства нарушаются, либо не может прийти к определенному вердикту, поскольку анализ модели требует слишком больших ресурсов.

Проверяемые свойства обычно разделяют на *свойства безопасности* (safety properties), означающие, что нечто нежелательное при любом варианте работы системы никогда не случается, и *свойства живучести* (liveness properties), означающие, наоборот, что что-то желательное рано или поздно произойдет. Иногда дополнительно выделяют *свойства стабильности* (или *сохранности*, persistence properties) — при

любом сценарии работы системы заданное утверждение в некоторый момент становится истинным и с тех пор остается выполненным — и *свойства справедливости* (fairness properties) — некоторое утверждение при любом сценарии работы будет выполнено в бесконечном множестве моментов времени. Те или иные свойства справедливости, например, что планировщик операционной системы после любого заданного момента времени гарантирует каждому активному процессу получение управления, часто являются исходными предположениями, при выполнении которых нужно проверить свойства безопасности или живучести.

Первые методы проверки моделей [170], предложенные в начале 1980-х, предполагали полное исследование моделей Крипке с помощью автоматического инструмента, для проверки формул логики ветвящегося времени CTL [171] был предложен достаточно эффективный алгоритм. Впоследствии были разработаны символические методы [172], в которых проверка выполняется за счет манипуляций с компактным представлением модели в виде упорядоченной двоичной разрешающей диаграммы (OBDD), что позволяет проверять модели с огромным (до $10^{200}$) количеством состояний. Однако не всякую систему можно представить достаточно компактным образом, например, размер OBDD автомата, описывающего умножение двоичных чисел, растет экспоненциально в зависимости от разрядности чисел.

Оказалось, что для некоторых видов временных логик проверочные алгоритмы неэффективны — для LTL [173] и CTL* [174] они линейны в зависимости от размера автомата, равного максимуму из числа состояний и числа переходов, но являются PSPACE-полными в зависимости от длины проверяемой формулы. Для μ-исчисления был найден алгоритм проверки моделей [175], сложность которого выражается формулой $O((S\Phi)^{d/2}T\Phi)$, где S — число состояний модели, Φ — длина проверяемой формулы, T — размер модели, а d — количество вложенных чередований операторов наибольшей и наименьшей неподвижной точки в Φ. Большинство исследований ведется в направлении поиска различных фрагментов μ-исчисления, в которых можно было бы использовать более эффективные алгоритмы.

Среди инструментов проверки моделей для верификации ПО наиболее широко используются следующие.

- SPIN [176,177], использующий в качестве моделей описанные на языке Promela системы взаимодействующих автоматов.
- Evaluator [178], входящий в состав набора инструментов CADP [179] для формального анализа протоколов и распределенных систем.
- SMV [180] и его развитие NuSMV [181] реализуют символическую проверку моделей.
- Design/CPN [182], использующий в качестве моделей размеченные сети Петри.
- UPPAAL [183] и Kronos [184], инструменты проверки моделей, используемые для верификации встроенных систем и систем реального времени. Оба основаны на расширенных временных автоматах, а Kronos еще позволяет использовать логику с явным временем для описания проверяемых свойств.
- Bogor [185], один из недавно появившихся инструментов, который, однако, активно используется в практических проектах.
- HyTech [186], инструмент для проверки гибридных моделей, используемых при создании встроенного ПО.

### 3.3.7. Методы и инструменты проверки согласованности

При проверке согласованности анализируется соответствие между двумя исполнимыми моделями, одна из которых моделирует проверяемый артефакт, обычно проект или реальную работу системы (ее компонента), а вторая — проверяемые свойства. Проверяемые свойства в этом случае — это требования к поведению системы или ее компонента, представленные в виде обобщенного автомата (системы переходов, сети Петри и пр.), все сценарии работы которого объявляются правильными.

В этом случае обычно проверяется, что все возможные сценарии поведения реализации возможны также и в спецификации. Иногда устанавливается их эквивалентность, т.е. дополнительно проверяется, что все сценарии поведения спецификации есть и у реализации.

Большинство методов и инструментов проверки согласованности используют для этого тестирование, и поэтому относятся к синтетическим методам верификации — к тестированию на основе моделей. Есть лишь несколько работ, в которых исследуются аналитические методы проверки согласованности, основанные на вычислении

специфической композиции реализационного и спецификационного автомата, выявляющей разность их поведений (см. [114], Глава 9, [135]). Из соответствующих инструментов можно упомянуть Verity-Check [187], а также BISIMULATOR и REDUCTOR, входящие в CADP [179]. Однако эти инструменты используются на практике в основном для верификации аппаратного обеспечения, а не ПО.

## 3.4. Динамические методы верификации

Динамические методы верификации используют результаты реальной работы проверяемой программной системы или ее прототипов, чтобы проверять соответствие этих результатов требованиям и проектным решениям.

Существует два основных вида динамических методов верификации: *мониторинг*, в рамках которого идет только наблюдение, запись и оценка результатов работы ПО при его обычном использовании, и *тестирование*, при котором проверяемое ПО выполняется в рамках заранее подготовленных сценариев. Во втором случае результаты работы тоже записываются, анализируются и оцениваются, основное отличие тестирования от мониторинга — целенаправленные попытки создать определенные ситуации, чтобы проверить работу ПО в них. Как видно, разделение мониторинга и тестирования несколько условно, тестирование всегда включает в себя и мониторинг. Общим для этих методов верификации является создание контролируемой среды выполнения ПО, обеспечивающей получение результатов его работы и измерение различных характеристик ПО, а также оценка этих результатов и характеристик.

Динамическая верификация может быть также реальной или имитационной, в зависимости от того, используется в ее ходе само ПО, его прототип или исполнимая модель.

Динамическую верификацию, служащую для обнаружения наличия ошибок и оценки качества ПО, следует отличать от *отладки* (debugging), основная задача которой — определение точного местоположения и исправление ошибок. Однако в ходе разработки динамическая верификация часто используется как часть отладки, и

поэтому, помимо самого факта наличия ошибок, должна давать как можно более детальную информацию об их локализации и нарушаемых ими ограничениях.

Основное достоинство динамических методов верификации — возможность получить информацию о реальной работе ПО и о реальных показателях его функциональности, производительности, надежности или переносимости. Имитационные методы этого качества лишены и применяются на практике либо для валидации проектных решений, либо для верификации моделей очень сложных систем, для которых эти модели представляют самостоятельную ценность, например, потому, что они впоследствии буду автоматически оттранслированы в исходный код.

С помощью динамических методов могут оцениваться все характеристики качества ПО за исключением удобства его сопровождения: функциональность, переносимость, производительность, надежность и удобство использования. Анализируемые атрибуты качества в первую очередь влияют на построение среды контролируемого выполнения программы для ее тестирования или мониторинга.

- При динамической верификации функциональности основное внимание уделяется протоколированию результатов работы операций, доступных элементов состояния компонентов системы, содержимого сообщений, которыми обмениваются компоненты системы, а также действительного порядка событий, насколько это позволяет сделать архитектура системы. При верификации систем реального времени также протоколируются временные интервалы между отдельными событиями.
- При контроле переносимости обычно нужно протоколировать такую же информацию, что и при контроле функциональности, или только ту ее часть, которая необходима для проверки одинаковости функционирования системы в разных окружениях.
- При анализе производительности протоколируются временные интервалы, в рамках которых система выполняет оцениваемые операции, записываются также объемы памяти (оперативной, в кэшах различных уровней, а также файла подкачки), занимаемые различными компонентами системы при выполнении этих операций, объем и интенсивность обмена данными по сети и пр. Хотя часть этих характеристик может использоваться и при анализе функциональности, в

общем случае измерения производительности нужно проводить отдельно, поскольку системы мониторинга функциональности обычно вносят в показатели производительности системы б*о*льшие искажения. Кроме того, поскольку один вызов часто выполняется за время, сравнимое с точностью измерения времени в компьютерных системах, для корректного определения времени работы отдельных операций часто нужно выполнять много одинаковых вызовов, чтобы получить среднее время их выполнения с небольшой ошибкой, что плохо совмещается с одновременным анализом функциональности.

- Для анализа надежности нужны статистические данные по количеству сбоев в системе за определенное время, времени ее доступности для пользователей, среднему времени восстановления при сбоях и пр. С одной стороны, собрать их проще, чем более детальную информацию о функциональности и производительности, но с другой стороны, от механизма ее сбора требуется повышенная надежность и способность корректно протоколировать сбои проверяемой системы.
- Динамическая верификация удобства использования практически не применяется, поскольку четко и непротиворечиво сформулировать адекватные требования к удобству использования ПО очень трудно. Чаще всего тестирование и мониторинг удобства использования выполняются для его валидации. Для этого привлекаются пользователи системы или посторонние лица, мало знакомые с системой, но достаточно хорошо знающие предметную область и задачи, для решения которых предназначена проверяемая система. Динамический контроль удобства использования требует специально организованного рабочего места, где было бы удобно протоколировать все действия пользователя, не вмешиваясь в них и никак не отвлекая его от работы с ПО. Обычно для этого используют специальное помещение с полупрозрачным стеклом в одной из стен, чтобы из-за него можно было наблюдать за действиями пользователя. Человек должен быть подготовлен, ему необходимо объяснить, что цель предстоящего мероприятия — оценка удобства системы, а не проверка его способностей и навыков, поэтому все его недоумения и «ошибочные» действия расцениваются как недостатки системы, а не его самого. Все операции

пользователя протоколируются (дополнительно к наблюдателям за полупрозрачным стеклом) с помощью видеокамеры, причем удобнее делать это так, чтобы вместе с экраном компьютера было видно лицо пользователя — так ему впоследствии гораздо легче вспомнить и объяснить суть возникавших проблем. Для этого часто используют зеркало, поставленное немного позади экрана компьютера так, чтобы стоящая за плечом пользователя камера снимала бы одновременно экран и выражение лица человека в зеркале.

Тестирование удобства использования мало отличается от мониторинга — в первом случае пользователю выдается список задач, которые он должен решить в рамках сеанса тестирования, а при мониторинге от него требуется просто разобраться, как работает программа, научиться с ее помощью делать то, что он обычно делает во время работы.

### 3.4.1. Мониторинг

При верификационном *мониторинге* (monitoring, runtime verification, online verification, passive testing) поведение проверяемой системы в ходе ее обычной работы протоколируется и оценивается его соответствие требованиям и проектным решениям. Частный случай мониторинга — *профилирование* (profiling), при котором обычно измеряются показатели производительности, однако довольно часто можно встретить употребление термина «профилирование» как для мониторинга, включающего контроль операций с памятью и взаимодействие параллельных потоков и процессов в системе.

Техники и инструменты мониторинга различаются по видам протоколируемой ими информации, способу получения данных о работе ПО и способу получения оценок характеристик ПО.

- Протоколируемая информация зависит от оцениваемых характеристик качества (см. выше) и от других целей проводимой верификации. Чаще всего записываются следующие данные.
    - Общие данные и метрики, связанные с проверяемыми характеристиками.

    - Факты вызовов операций, отправки и получения сообщений, обращений к синхронизационным примитивам (семафорам, мьютексам, барьерам и т. д.).
    - Значения параметров вызовов операций и их результатов.
    - Содержимое сообщений, передаваемых между системой и окружением и между компонентами системы.
    - Значения доступных внутренних переменных.
    - Временные метки отдельных событий или интервалы между событиями.
    - Время работы отдельных операций или, реже, участков кода.
    - Объем и количество захватываемой и освобождаемой динамической памяти.
    - Распределение используемой памяти по объектам и их типам.
    - Использование кэшей различных уровней, промахи кэшей.
    - Использование файлов подкачки.
    - Объем и количество передаваемых сообщений различных типов.
  - Данные, анализируя которые можно выявлять типовые ошибки и проблемные ситуации.
    - Обращения к динамической памяти и указателям. Используются для выявления ошибки при работе с динамической памятью: использования неинициализированных объектов, выхода за границы массивов и выделенных блоков памяти, нарушений парности операций захвата и освобождения памяти, утечек памяти за счет «потери» указателей и пр.
    - Обращения к синхронизационным примитивам и общим ресурсам параллельных процессов и потоков. Позволяют выявить ошибки взаимодействия при параллелизме: *ситуации гонок* (race conditions), в которых результаты работы системы зависят от очередности обращения процессов к общим данным, и *тупиковые ситуации* (deadlocks), в которых процессы не могут продолжать

работу, потому что ждут друг от друга освобождения нужных им ресурсов, и т.п.

        - Статистика сетевых взаимодействий позволяет обнаружить возможные атаки и нарушения политик сетевой безопасности по определенным шаблонам взаимодействий и необычным видам сетевой активности системы.

- По способу получения данных о работе ПО техники мониторинга делятся на использующие инструментирование исходного или бинарного кода и симуляторные.
    - Выделяют следующие виды техник инструментирования.
        - *Ручное* (manual). Разработчик сам вносит в нужные ему места обращения к библиотечным функциям системы мониторинга.
        - *Компиляторное* (compiler assisted). Вызовы библиотеки мониторинга расставляются компилятором в специальном режиме компиляции.
        - *На основе бинарной трансляции* (binary translation). Бинарный код ПО подвергается обработке специализированным инструментом, вставляющим обращения к библиотеке мониторинга.
        - *Инструментирование времени выполнения* или *инъекция времени выполнения* (runtime instrumentation, runtime injection). Проводится инструментом, выполняющим ПО при мониторинге его поведения. При инъекции инструмент вставляет только минимально необходимые переходы на функции мониторинга, общий объем вставок несколько меньше.
    - *Симуляторный мониторинг* (simulation-based, hypervisor-based) основан на протоколировании работы проверяемого ПО симулятором, на котором выполняется. Существуют аппаратные решения, поддерживающие выполнение симуляторного мониторинга.
- По способу получения оценок характеристик ПО техники мониторинга делятся на основанные на событиях (event based) и статистические (statistical).

- *Техники, основанные на событиях*, используют для получения метрик полную запись событий и их атрибутов (времени, содержания, задействованных объемов памяти и кэшей различных типов и пр.).
- *Статистические техники* мониторинга оценивают показатели работы системы, основываясь на наборе мгновенных снимков ее состояния, получаемых через случайные или одинаковые интервалы времени. Статистические техники менее аккуратны и точны, но зато меньше нагружают работающую систему и вносят менее заметные искажения в ее поведение.

Инструменты мониторинга достаточно разнообразны, они предназначены обычно для контроля лишь одного атрибута качества, например, производительности (см. список инструментов в Wikipedia [188]), корректности взаимодействия параллельных процессов [189], защищенности ПО и сетей [190,191] и пр.

Наиболее широко используемыми инструментами мониторинга являются профилировщики, входящие в состав инструментов разработки ПО (например, gprof, входящий в пакет GNU binutils [192], сопровождающий компилятор gcc [193] или Visual Studio Team System Profiler [194], интегрированный с Microsoft Visual Studio). Из распространяемых отдельно профилировщиков можно отметить свободно распространяемый Valgrind [195,196], коммерческие Rational PurifyPlus [197] и Intel VTune Performance Analyzer [198].

### 3.4.2. Тестирование

*Тестирование* (testing) является методом верификации, в рамках которого результаты работы тестируемой системы или компонента в ситуациях из выделенного конечного набора проверяются на соответствие проектным решениям, требованиям, общим задачам проекта, в рамках которого эта система разрабатывается или сопровождается. Ситуации, в которых выполняется тестирование, называют *тестовыми ситуациями* (test situations, test purposes), а процедуры, описывающие процесс создания этих ситуаций и проверки, которые необходимо выполнить над полученными результатами, — *тестами*.

Часто говорят, что тестирование должно проверять соответствие работы ПО требованиям к ней [13,199], однако «требования» здесь нужно понимать очень широко, в частности, включать в них детали проектных решений, определяемых на достаточно поздних этапах проекта. Довольно часто проводят тестирование, в котором проверяют только то, что система не «падает», не создает исключительных ситуаций или «странных», совершенно неожиданных для разработчиков результатов. Все эти ограничения обычно можно считать несформулированной явно, но подразумеваемой частью требований (стоит помнить, однако, что системы некоторых типов в определенных ситуациях *должны* «падать» или создавать исключительные ситуации). Кроме того, чем меньше при тестировании задействованы достаточно точные требования к тому, что система должна делать, чем более оно ориентировано лишь на поиск сбоев или других видов инцидентов, тем меньше пользы оно приносит проекту. Поэтому такое тестирование оправдано только как прелюдия к серьезному тестированию, для проверки того, что система в целом работоспособна, или в тех ситуациях, где описание требований вообще отсутствует, или же они сформулированы крайне нечетко и неполно.

Тестирование может использоваться и для валидации, в этом случае оно проверяет соответствие поведения ПО ожиданиям и потребностям пользователей и заказчиков, а для оценки этого соответствия должны привлекаться сами пользователи, их представители, бизнес-аналитики или эксперты в предметной области.

Тестирование, как и верификация вообще, служит для поиска ошибок или дефектов и для оценки качества ПО. Эффективность решения обеих этих задач во многом определяется тем, какой именно набор тестовых ситуаций выбран для проведения тестирования. Чтобы иметь некоторые гарантии аккуратности полученных в ходе тестирования оценок качества, необходимо выбирать тестовые ситуации систематическим образом, в соответствии с основными задачами и рисками проекта. Правила, определяющие набор необходимых тестовых ситуаций, называют *критериями полноты (*или *адекватности) тестирования* (test adequacy criteria) [200-202]. Обычно такой критерий использует разбиение всех возможных при работе проверяемого ПО ситуаций на некоторые классы эквивалентности, такие, что ситуации из одного класса достаточно похожи друг на друга и работа ПО в них не должна

отличаться сколь-нибудь значительным образом. Полноту тестирования при этом можно определять по проценту задействованных, «покрытых» в нем классов ситуаций. Такой критерий полноты называется *критерием тестового покрытия* (test coverage criterion), а процент покрытых в результате работы тестов классов ситуаций — *достигнутым тестовым покрытием*.

Заметим, что различных критериев полноты довольно много (см. ниже), а тестирование, покрывшее 100% выделенных по одному из таких критериев классов ситуаций, и поэтому часто называемое *полным*, в действительности не является полным или исчерпывающим в каком-либо смысле, связанным со строгими гарантиями отсутствия ошибок и соответствия ПО требованиям. Такого рода исчерпывающее тестирование невозможно для практически значимых систем.

Подготовка и проведение тестирования в проекте создания или сопровождения ПО проходят примерно по следующему плану.

- Определение целей тестирования (test objectives, test goals) на основе задач и рисков проекта. Такие цели очерчивают проверяемые характеристики и свойства ПО, тщательность тестирования отдельных компонентов и подсистем. Они также определяют используемые в проекте виды тестирования и методы построения тестов.
- Определение требований, свойств и ограничений, которые должны проверяться в ходе тестирования.
- Определение критерия полноты тестирования, который будет использоваться в данном проекте. Критерий полноты должен быть согласован с целями тестирования, он управляет выбором тестовых ситуаций для тестирования, а также определяет, когда можно прекратить тестирование.
- Построение набора тестов, нацеленных на достижение выбранного критерия полноты и проверяющих определенные ранее требования и ограничения.
- Отладка, выполнение тестов и получение итоговых результатов тестирования в виде сообщений о выполненных действиях и нарушениях проверяемых ограничений. Отладка тестов включает пробные прогоны, устранение обнаруживаемых ошибок в самих тестах, а также описаниях проектных решений и требований, на основе которых получены тесты. По результатам отладки

может потребоваться пополнение набора тестов для достижения максимального возможной полноты тестирования в рамках выделенных ресурсов.

- Анализ результатов тестирования, составление отчетов о найденных дефектах и о полученных оценках качества ПО.

### 3.4.3. Виды тестирования

Классификация видов тестирования достаточно сложна, потому что может проводиться по нескольким разным аспектам.

- По уровню или масштабу проверяемых элементов системы тестирование делится на следующие виды.
  - *Модульное* или *компонентное* (unit testing, component testing) — проверка корректности работы отдельных компонентов системы, выполнения ими своих функций и предполагаемых проектом характеристик.
  - *Интеграционное* (integration testing) — проверка корректности взаимодействий внутри отдельных групп компонентов.
  - *Системное* (system testing) — проверка работы системы в целом, выполнения ею своих основных функций, с использованием определенных ресурсов, в окружении с заданными характеристиками.
- По проверяемым характеристикам качества тестирование может быть тестирование функциональности, производительности (и по времени, и по другим ресурсам), надежности, переносимости или удобства использования. Более специфические виды тестирования, нацеленные на оценку отдельных атрибутов, — тестирование защищенности, совместимости или восстановления при сбоях.

  Специфическим видом тестирования, нацеленным на минимизацию риска того, что в результате доработки или внесения ошибок качество системы изменилось в худшую сторону, является *регрессионное тестирование* (regression testing). При его проведении используется уже применявшийся ранее набор тестов, и оно должно выявить различия между результатами, полученными на этих тестах ранее, и наблюдаемыми после внесения изменений.

- По источникам данных, используемых для построения тестов, тестирование относится к одному из следующих видов.
  - *Тестирование черного ящика* (black-box testing, часто также называется *тестированием соответствия*, conformance testing, или *функциональным тестированием*, functional testing) — нацелено на проверку соблюдения требований. Использует критерии полноты, основанные на требованиях, и техники построения тестов, использующие только информацию, заданную в требованиях к проверяемой системе.

    Частными случаями этого вида тестирования являются тестирование на соответствие стандартам и *квалификационное* или *сертификационное тестирование*, нацеленное на получение некоторого сертификата соответствия определенным требованиям или стандартам.
  - *Тестирование белого ящика* (white-box testing, glass-box testing, также *структурное тестирование*, structural testing) [201] — нацелено на проверку корректности работы кода. Использует критерии полноты и техники построения тестов, основанные на структуре проверяемой системы, ее исходного кода.
  - *Тестирование серого ящика* (grey-box testing) использует для построения тестов как информацию о требованиях, так и коде. На практике оно встречается чаще, чем предыдущие крайние случаи.
  - *Тестирование, нацеленное на ошибки* — использует для построения тестов гипотезы о возможных или типичных ошибках в ПО такого же типа, как проверяемое. К этому типу относятся, например, следующие виды тестирования.
    - *Тестирование работоспособности* (sanity testing, smoke testing), нацеленное на проверку того, в систему включены все ее компоненты и операции, и система не дает сбоев при выполнении своих основных функций в простейших сценариях использования.
    - *Тестирование на отказ*, пытающееся найти ошибки в ПО, связанные с контролем корректности входных данных.

    - *Нагрузочное тестирование* (load testing), проверяющее работоспособность ПО при больших нагрузках — больших объемах входных, выходных или промежуточных данных, большой сложности решаемых задач, большом количестве пользователей, работающих с ПО и пр.
    - *Тестирование в предельных режимах* (stress testing), проверяющее работоспособность ПО на границах его возможностей и на границах той области, где оно должно использоваться.

- По роли команды, выполняющей тестирование, оно может относиться к следующим видам.
  - *Внутреннее тестирование* выполняется в рамках проекта по разработке системы силами организации-разработчика ПО.
  - *Независимое тестирование* выполняется третьими лицами (не разработчиками, не заказчиками и не пользователями) для получения объективных и аккуратных оценок качества системы.
  - *Аттестационное тестирование* (приемочные испытания) выполняется представителями заказчика непосредственно перед приемкой системы в эксплуатацию для проверки того, что основные функции системы реализованы. Обычно аттестационные тесты являются очень простыми тестами функциональности и производительности.
  - *Пользовательское тестирование* осуществляется силами пользователей системы. У него есть два часто упоминаемых частных случая.
    - *Альфа-тестирование* (alpha-testing) выполняется самими разработчиками, но в среде, максимально приближенной к рабочему окружению системы и на наиболее вероятных сценариях ее реального использования.
    - *Бета-тестирование* (beta-testing) выполняется пользователями, желающими познакомиться с возможностями системы до ее официального выпуска и передачи в эксплуатацию.

### 3.4.4. Критерии полноты тестирования

Критерии полноты тестирования должны выбираться на основе рисков проекта и того, какие ошибки в тестируемом ПО наиболее важны. На практике используют критерии следующих типов.

- Критерии покрытия структурных элементов тестируемой системы, которые выполняются или задействуются в ходе тестов. Такие критерии называются *структурными критериями* и используются, если код тестируемой системы доступен для анализа. Структурные критерии покрытия позволяют отслеживать, были ли отдельные элементы кода хотя бы раз выполнены в ходе тестирования, однако не могут ничем помочь в обнаружении нереализованных требований.
    - На уровне отдельных операций используют критерии покрытия кода [200,201], основанные на элементах графа потока управления или графа потоков данных. К первому типу относятся критерий покрытия инструкций или ветвлений в коде, критерий покрытия комбинаций элементарных условий, используемых в ветвлениях. Критерий покрытия использований переменных или критерий покрытия путей от точек присваивания переменной некоторого значения к точкам ее использования относятся к критериям покрытия потоков данных.
    - На уровне компонента или группы компонентов используют покрытие элементов графа вызовов их операций или вхождений операций чтения и записи отдельных элементов данных этих компонентов в код тестируемых операций.
    - На уровне системы в целом или крупных подсистем используют критерии покрытия возможных сценариев взаимодействия компонентов.
- Специальным частным случаем структурных критериев являются *критерии покрытия, основанные на структуре входных данных* тестируемых операций. Эти критерии полезны в тех случаях, когда код системы для анализа недоступен, или описание требований дает недостаточно информации для построения аккуратных тестов.
- Критерии покрытия элементов требований. Они называются *функциональными критериями* и устроены подобно структурным критериям покрытия, только в

качестве покрываемых элементов рассматриваются не сценарии взаимодействия компонентов, участки их кода или комбинации условий ветвлений, а сценарии использования, отдельные правила и ограничения в требованиях и комбинации условий, при которых задействуются эти правила и ограничения.

Такие критерии полноты позволяют отслеживать, что тестирование проверяет все ограничения, указанные в требованиях, однако они не позволяют целенаправленно обнаружить дополнительные функции или закладки в коде, что можно сделать, используя структурные критерии.

- *Критерии полноты на основе гипотез об ошибках*. Такие критерии строятся на базе некоторой схемы возможных ошибок в проверяемом ПО. Эта схема создается на основе рисков проекта, опыта разработчиков и тестировщиков, а также типичных ошибок, возможных в системах данного типа. Часто используются таксономии ошибок, подобные [53,203], — это позволяет с минимальными усилиями значительно снизить риск не обнаружить важную для пользователей ошибку.

  Один из видов таких критериев — критерий покрытия мутантов [204]. Мутантом называется проверяемая программа с небольшим внесенным в нее изменением (заменой константы 0 на 1, оператора + на -, знака < на > или <=, перемены местами ветвей в операторе if…else и т.п.), отличающаяся по поведению от исходной. Созданы наборы операторов мутаций для различных языков программирования [205-207], позволяющие получить систематичные семейства мутантов. Полнота тестов измеряется процентом мутантов, которые они выявляют.

- *Критерии покрытия элементов моделей поведения* ПО являются самым общим типом критериев полноты. Перечисленные выше критерии соответствуют наиболее широко используемым моделям. Более специфические модели используются в тех случаях, когда перечисленные выше виды критериев не удается применить или они не дают достаточно низкого уровня риска пропустить важную ошибку в проверяемом ПО. Примерами таких критериев могут служить критерий покрытия состояний и критерий покрытия переходов в модели ПО в виде конечного автомата.

- *Смешанные критерии*. Все указанные выше виды критериев покрытия имеют как достоинства, так и недостатки. Чтобы преодолеть последние на практике обычно применяют комбинацию из критериев разных видов. Часто используют сочетание функционального и структурного критериев — первый обеспечивает отслеживание покрытия требований, второй — покрытия структуры системы и гарантии обнаружения участков кода с неясной функциональностью.

### 3.4.5. Техники построения тестов

Критерии полноты тестирования, описанные выше, позволяют оценить систематичность и полноту (в некотором смысле) набора тестов. Чтобы получить этот набор тестов, нужно использовать специальные *техники построения тестов* (test selection techniques).

Прежде, чем обсуждать эти техники, стоит рассмотреть общую структуру теста. Как уже говорилось, тест представляет собой процедуру, обеспечивающую создание некоторой специфической ситуации с точки зрения поведения тестируемой системы и проверку правильности работы системы в этой ситуации. Соответственно, при построении теста, нужно выбрать нужную ситуацию, определить способ ее создания и определить процедуру проверки правильности работы тестируемой системы.

Тестовая ситуация обычно включает в себя следующие элементы.

- Оказываемое на систему воздействие с некоторыми данными, результаты которого надо проверить в первую очередь.

  Это может быть вызов операции с определенными аргументами, посылка сообщения определенного содержания, нажатие на кнопку диалога после того, как другие поля этого диалога заполнены некоторыми данными, выполнение команды в командной строке с некоторыми опциями и аргументами и т.п.

- Внутреннее состояние тестируемой системы.

  Поскольку состояние чаще всего невозможно установить напрямую, для его достижения нужно использовать другие воздействия на систему. При этом их результаты тоже необходимо проверять, так как само состояние чаще всего недоступно для прямого наблюдения.

  В результате достижение определенной ситуации чаще всего требует

использования *тестовой последовательности* (test sequence) — последовательности обращений к системе с определенными данными.

- Иногда на работу систему влияют внешние условия, воспринимаемые системой, помимо оказываемых на нее воздействий. Это могут быть измеряемые автоматически физические показатели окружающей среды или результаты взаимодействий с другими системами. Часто такие условия можно промоделировать программно, задавая определенную конфигурацию системы, имитируя деятельность других систем или заменив датчики физических показателей управляемыми модулями. В тех случаях, когда это сделать нельзя, нужно создавать модель рабочего окружения системы, позволяющую изменять внешние условия при тестировании.

Для выбора (или построения) тестовых ситуаций и составления тестового набора используют техники следующих типов.

- *Вероятностное тестирование* (random testing) [208]. При использовании этого подхода каждая возможная тестовая ситуация описывается некоторым набором параметров, все значения которых генерируются как псевдослучайные данные с определенными распределениями. Проще всего таким способом получать тестовые наборы для систем без внутреннего состояния, при этом достаточно задать распределения вероятностей значений различных параметров. Для систем с состоянием используются описания распределения вероятностей возможных сценариев работы с ними в виде марковских цепей [209,210].

  Исходной информацией для задания распределений значений параметров является частота их использования при реальной работе системы (*профиль использования*), величина риска, связанного с ошибками в системе в соответствующей ситуации. Чаще всего, однако, используется вероятностное тестирование с равномерными распределениями, просто потому, что оно является наиболее дешевым способом получения большого количества тестов. Стоит помнить, однако, что вероятностные тесты дают оценку качества ПО неизвестной достоверности — иногда они действительно систематически проверяют различные возможности системы, иногда нет, но чтобы узнать это, нужно использовать другой способ оценки качества. Они также практически

неспособны находить сложные ошибки, возникающие из-за сочетания различных факторов.

- *Тестирование на основе классов эквивалентности* (partition testing, см., например, [211], а также [212,213]). Для выбора тестов по этой технике все возможные ситуации разбиваются на конечное множество классов эквивалентности. Обычно это делается так, чтобы различия в поведении тестируемого ПО в эквивалентных ситуациях были несущественны, а в неэквивалентных — достаточно велики. Часто за основу разбиения выбирается используемый критерий покрытия — ситуации, которые соответствуют одному покрываемому элементу в рамках этого критерия, считают эквивалентными. Далее тесты строятся так, чтобы в каждом классе эквивалентности был хотя бы один тест. Пример такой техники — метод функциональных диаграмм из книги Майерса [213].

  Техники такого рода можно использовать в широкой области, их можно нацелить на определенные ошибки, даже достаточно сложные, но они требуют участия человека, построение тестов с их помощью тяжело автоматизировать.

- *Комбинаторное тестирование* [214]. В этом случае произвольная тестовая ситуация также описывается набором параметров, каждый из которых может принимать только конечное множество значений. Ситуации для тестирования выбираются таким образом, чтобы реализовать определенные комбинации значений параметров, например, это могут быть вообще все комбинации, или комбинации всех пар значений, или два параметра с одинаковыми множествами значений должны принимать равные значения, а остальные — произвольные, и т.п. Для получения конечного множества значений параметра обычно используют выделение классов эквивалентности этих значений.

  Эти техники обеспечивают более систематичное тестирование, чем вероятностные, но несколько более трудоемки в использовании, хотя и проще, чем тестирование на основе классов эквивалентности. Стоит отметить, что при росте количества различных значений параметров количество их различных комбинаций в комбинаторных тестах может расти экспоненциально.

- *Сценарное тестирование* (scenario-based testing, см., например, [215]). Тесты строятся на основе сценариев использования системы, сценариев ее взаимодействия с другими системами или сценариев взаимодействия ее компонентов друг с другом. Такие сценарии классифицируются, и для каждого выделенного типа сценариев создается тест, повторяющий общую структуру таких сценариев.

  Сценарное тестирование позволяет сэкономить усилия при разработке тестов на основе требований, представленных в виде вариантов использования. В зависимости от используемой схемы классификации сценариев оно может быть поверхностным или достаточно полным. Автоматизируется такое тестирование с трудом, поскольку обычно описываемые в требованиях сценарии использования нельзя прямо применять как тестовые сценарии — для этого необходимо их обобщить, ввести некоторые дополнительные параметры, добавить возможности разветвлений и определить процедуры проверки корректности получаемых на разных шагах результатов.
- *Тестирование, нацеленное на определенные ошибки*. (fault-based testing, risk-based testing). С помощью таких техник строят тесты, прямо нацеленные на обнаружение ошибок некоторого типа. Они достаточно часто используются на практике, поскольку позволяют существенно снижать риски проекта. Возможность автоматизации определяется видом ошибок, на которые нацеливаются тесты.

  Примером такой техники является *тестирование граничных значений* (boundary testing), в рамках которых в качестве данных для тестов используются значения, расположенные на границах областей, в которых тестируемая операция ведет себя по-разному.
- *Автоматное тестирование* (см., например, [73]). В рамках техник автоматного тестирования тесты строятся как пути на графе переходов автоматной модели, описывающей требования к поведению или проект тестируемого ПО. Поскольку они существенно используют формальные модели ПО, более подробно техники такого рода описаны в разделе про тестирование на основе моделей.

- *Смешанные техники*. В большинстве случаев на практике используют комбинацию из техник нескольких типов, поскольку ни один из них не покрывает полностью область применения и достоинства ни одного другого. Кроме этого, часто применяется *адаптивное тестирование* (adaptive testing), которое опирается на получаемую в ходе выполнения тестов информацию для выявления наиболее проблемных частей системы и построения новых тестов, более прямо нацеленных на вероятные ошибки. Одной из техник такого рода является *исследовательское тестирование* (exploratory testing) [216]. Оно дополнительно делает упор на получение человеком более полной информации о свойствах ПО в процессе выполнения тестов и его способности отмечать странности в работе системы, которые можно использовать для эффективного выявления более существенных ошибок.

  Примером комбинации техник вероятностного тестирования и нацеливания на некоторые классы ситуаций является техника генерации структурных тестов на основе генетических алгоритмов [217]. В ее рамках исходный набор тесов генерируется случайно, после чего запускается генетический алгоритм, отбирающий такие тесты, который обеспечивает максимальное покрытие реализации тестируемых операций. Тест считается тем «лучше» с точки зрения дальнейшего отбора, чем ближе его выполнение подходит к еще не покрытым элементам кода.

Помимо выбора набора ситуаций для тестирования и определения способа их достижения, необходимо определить процедуру проверки получаемых при тестировании результатов на корректность. Такая процедура называется *тестовым оракулом* (test oracle). Существуют следующие способы организации оракулов.

- При неавтоматизированном тестировании роль оракула играет человек, выполняющий тесты. Именно он оценивает, насколько получаемые результаты соответствуют требованиям, проектным решениям, задачам проекта, нуждам пользователей.
- Очень часто при автоматизированном тестировании проверяется отсутствие сбоев и исключительных ситуаций. Это частичный оракул, который выражает только малую часть требований к ПО.

- При автоматизации выполнения тестов иногда используют оракул, организованный в виде проверки равенства полученных в тесте результатов ожидаемым, вычисленным заранее. Ожидаемые результаты определяют для используемых тестовых ситуаций при помощи более детального анализа требований и интервью с пользователями или экспертами в предметной области.
- Иногда тестовый оракул сравнивает полученные в тесте результаты с вычисляемыми другой реализацией тех же функций. В качестве другой реализации могут использоваться другие системы с похожей функциональностью, более ранние версии тестируемой системы, прототипы и исполнимые модели, созданные на более ранних этапах проекта. Ситуации, в которых между тестируемой системой и другой реализацией возникают разногласия, проверяются особо, поскольку ошибка в них может быть с любой стороны. Если есть несколько других реализаций, для определения правильного результата может использоваться процедура голосования между ними.
- Если проверяемая функция имеет легко вычисляемую обратную, ее можно использовать для построения оракула — вычислить обратную функцию от результатов и сравнить полученные данные с входными данными проверяемой функции.
- Часто известны некоторые свойства результатов, которые заведомо должны быть выполнены, свойства значений переменных проверяемой программы в определенных точках кода или инварианты внутренних данных тестируемого компонента. Оракул может проверять эти свойства и инварианты в ходе тестирования.
- Наиболее полно проверить корректность поведения тестируемой системы можно с помощью оракула, получаемого из формальной модели требований и проектных решений [218]. Такие оракулы используются в тестировании на основе моделей.

При использовании оракулов в автоматизированном тестировании стоит помнить, что практически всегда оракул может проверить лишь часть имеющихся требований и правил. Для более аккуратной оценки качества ПО необходимо

достаточно четко представлять границы этой части и планировать определенные усилия для проверки оставшихся ограничений.

### 3.4.6. Инструменты автоматизации тестирования

Инструменты автоматизации тестирования делятся на следующие классы.

- *Инструменты управления информацией о тестах* (test management tools) [219]. Эти инструменты собирают данные о тестах и их связях с другими артефактами разработки, предоставляя доступ к ней тестировщикам, инженерам по качеству, руководству проектов. Среди них наиболее известны TestManager [220] от IBM/Rational и TestDirector [221] от HP/Mercury.
- *Инструменты мониторинга*, используемые при тестировании для протоколирования работы тестируемой системы. См. раздел о мониторинге.
- *Инструменты сбора данных о тестовом покрытии* (test coverage tools) [222]. Такие инструменты позволяют измерять достигнутое при тестировании покрытие кода, обычно по критериям покрытия инструкций и/или ветвлений.
- *Каркасы выполнения тестов* (test execution frameworks). В рамках такого каркаса тесты запускаются как исполнимые модули или оформляются как программы, использующие API каркаса для мониторинга их работы. Помимо автоматизации запуска тестов часто предоставляются возможности по оценке того, выполнился тест успешно или нет, и дополнительные библиотеки для организации проверок в тестах и сброса трассировочной информации.

  Наиболее известными инструментами такого типа являются инструменты для автоматизации модульного тестирования семейства xUnit (jUnit и его производные, см. их список в Wikipedia [223], а также [224]) и TET [225].
- *Инструменты генерации тестовых данных* (test input generators). Такие инструменты, в свою очередь, разбиваются на следующие группы.
  - *Вероятностные генераторы* используют генерацию псевдослучайных данных.
  - *Комбинаторные генераторы* используют комбинаторные техники создания тестов, чаще всего основанные на построении *покрывающих наборов* (covering arrays) [226-228].

- *Генераторы сложных данных*, в виде заполнений баз данных, XML-документов или текстов на языках, описываемых некоторыми грамматиками. Таких инструментов очень много, инструменты генерации заполнений баз данных чаще всего коммерческие, см. [229-232], инструменты для генерации XML-документов обычно свободные или исследовательские [233-237]. Инструменты генерации тестовых данных на основе грамматик известны с 1970-х годов [238], но есть и несколько более современных таких инструментов [239]. В ИСП РАН разработаны инструмент SynTESK [240], генерирующий по грамматике языка набор тестовых программ для синтаксических анализаторов, а также каркасы для создания генераторов сложных данных OTK [241] и Pinery [242].

- Инструменты доступа к специализированным интерфейсам позволяют работать с этими интерфейсами в тестах с помощью обращений к программному интерфейсу.
  - *Инструменты тестирования пользовательского интерфейса* (GUI testing tools) [243,244]. Такие инструменты чаще всего основаны на записи действий пользователя (заполнения полей форм, нажатия кнопок, выбора пунктов меню и пр.) и возможности их воспроизведения. Обычно поддерживаются также возможности изменения редактирования выполнения тестов, записываемых на определенных языках, и использования различных данных в качестве заполнения полей.

    Эти инструменты обычно нацелены на проверку работы интерфейсов определенного типа — Windows GUI, GUI библиотек KDE или Gnome для Linux, WebUI [244]. Наиболее известны из них IBM/Rational Robot [245], HP/Mercury QuickTest Professional [246], Empirix e-Tester [247].
  - Специализированные *инструменты тестирования протоколов* [248] предоставляют поддержку работы через программный интерфейс с определенными телекоммуникационными или прикладными протоколами, а также часто и проверку корректности обмена сообщениями в их рамках.

- Инструменты автоматизации построения тестов на основе моделей позволяют автоматизировать выбор тестовых ситуаций, создание оракулов и оценку полноты тестирования. Подробнее они рассматриваются в разделе, посвященном тестированию на основе моделей.

Многие используемые на практике инструменты сочетают в себе функции нескольких из указанных типов, поскольку тестировщикам обычно удобнее пользоваться одним инструментом, чем несколькими.

## 3.5. Синтетические методы

Синтетические методы верификации сочетают техники нескольких типов — статический анализ, формальный анализ свойств ПО, тестирование. Некоторые из таких методов породили в последние 10-15 лет самостоятельные бурно развивающиеся области исследований, в первую очередь, *тестирование на основе моделей* и *мониторинг формальных свойств* (runtime verification, passive testing).

### 3.5.1. Тестирование на основе моделей

*Тестирование на основе моделей* (model based testing) использует для построения тестов формальные модели требований к ПО и принятых проектных решений. Как критерии полноты тестирования, так и оракулы строятся на основе информации, содержащейся в этих моделях. Получаемые в результате тесты обычно слабо связаны со специфическими особенностями кода тестируемой системы, но содержат представительный набор ситуаций с точки зрения исходной модели.

Истоки методологии тестирования на основе моделей лежат в проведенных в 1950-х годах исследованиях возможности определения структуры конечного автомата, моделирующего поведение данной системы, на основе результатов экспериментов с этой системой [249]. Впоследствии были разработаны методы [250,251], позволяющие на основе модели в виде конечного автомата строить наборы тестов, успешное выполнение которых при определенных ограничениях на проверяемую систему гарантирует эквивалентность поведения этой системы заданной модели.

В конце 1980-х годов интерес к тестированию на основе моделей возобновляется, и начинают разрабатываться новые подходы, использующие другие виды моделей, помимо конечных автоматов. Основной области их применения в то время было тестирование реализаций телекоммуникационных протоколов на соответствие стандартам этих протоколов. Примерно в это время разрабатывается стандарт ISO 9646 [252] на такое тестирование, учитывающий возможность использования формальных моделей. С середины 1990-х годов тестирование на основе моделей начинает использоваться для других видов программных систем за счет использования более хорошо масштабируемых моделей в виде программных контрактов [253,254].

В настоящее время методы тестирования на основе моделей используют следующие типы моделей и техник.

- *Методы проверки согласованности автоматов и систем переходов.* Такие методы относятся к одному из трех типов, в зависимости от используемых моделей.
  - Обычные и расширенные конечные автоматы [73,255]. Методы построения тестов на основе конечных автоматов наиболее глубоко разработаны, известны их точные ограничения и гарантии полноты выполняемых проверок. Методы, использующие расширенные автоматы, сводят их к обычным, но применяют более детальные критерии покрытия, основанные на использовании данных в расширенных автоматах.

    Наиболее известными инструментами создания тестов на основе таких моделей являются BZ-TT [255,256], использующий модели, описанные на языке B [257], и Gotcha-TCBeans [258], использующий язык Murφ [259] или UML Statecharts в рамках набора инструментов AGEDIS [260].
  - Системы переходов [73]. Такие методы чаще используются при тестировании распределенных систем, поскольку моделирование таких систем с помощью конечных автоматов очень трудоемко. Большинство этих методов не определяют практически применимых критериев полноты и не дают конечных тестовых наборов для реальных систем,

поэтому использующие их инструменты опираются на те или иные эвристики для обеспечения конечности набора тестов.

Первые методы построения тестов по LTS-моделям были разработаны в работах Бринксмы [261] и Тритманса [262]. Наиболее известны из инструментов, созданных на основе результатов Тритманса, TorX [263] и TGV [264]. Последний инструмент входит в набор инструментов верификации CADP [179].

Помимо обычных систем переходов используются и временные (расширенные с помощью таймеров), построение тестов на их основе возможно с использованием инструмента UPAAL [183].

  - Программные контракты. В методах такого рода описание требований к тестируемой системе представляет собой набор программных контрактов, иногда с дополнительным определением явного преобразования состояния системы. Тесты строятся на основе автоматных моделей, являющихся абстракциями от этих контрактов. Примеры инструментальной поддержки таких методов — инструменты технологии UniTESK [265,266], созданной в ИСП РАН, и инструмент SpecExplorer [267], разработанный в Microsoft Research.

Более подробные обзоры инструментов, поддерживающих такие методы см. в [73,268].

- *Методы построения тестов на основе формального анализа свойств ПО* используют формальный анализ для классификации тестовых ситуаций и нацеленной генерации тестов.
  - Методы на основе проверки моделей [269-271]. В рамках таких методов тестовые ситуации выбираются как представители классов эквивалентности, задаваемых критерием покрытия. Соответствующая ситуации тестовая последовательность строится как контрпример при проверке модели (model checking) на выполнение свойства, являющегося отрицанием условия достижения этой ситуации.

- Методы на основе дедуктивного анализа (например, [272]). В этих методах выбираемые тестовые ситуации соответствуют особым случаям в дедуктивном анализе свойств тестируемой системы.

- *Методы построения тестов с помощью символического выполнения* (symbolic execution). Такие методы используют символическое описание проходимого во время выполнения теста пути по коду программы (или формальных проверяемых спецификаций) в виде набора предикатов. Это описание позволяет выбирать новые тестовые ситуации так, чтобы они покрывали другие пути и строить тесты с помощью техник *разрешения ограничений* (constraint solving) [273,274]. Символическое выполнение в комбинации с конкретизацией тестовых данных используется и для построения тестов, нацеленных на типичные дефекты, такие, как использование неинициализированных объектов, тупики и гонки параллельных потоков [275].

Более детальную таксономию методов и инструментов тестирования на основе моделей можно найти в [276].

Можно отметить, что наиболее развиты методы тестирования на основе моделей первой из перечисленных выше групп. Поддерживающие их инструменты все шире начинают использоваться в промышленных проектах. Методы построения тестов на основе проверки моделей и на основе символического выполнения активно развиваются и начинают воплощаться в инструменты (см. далее о синтетических методах построения тестов).

### 3.5.2. Мониторинг формальных свойств ПО

*Мониторинг формальных свойств ПО* (для этой области используется два английских термина — runtime verification и passive testing) стал развиваться как отдельное направление исследований в конце 1990-х годов. В основе этого подхода лежит фиксация формальных свойств ПО, которые необходимо проверить и встраивание их проверок в систему мониторинга. В дальнейшем выполняется мониторинг ПО, в ходе которого контролируются и выделенные свойства.

В публикациях упоминаются следующие методы мониторинга формальных свойств ПО.

- *Использующие описание свойств с помощью обычных и временных логик* (см. [277,278]). Подобные техники осуществляют контроль свойств, записанных в виде формул временных логик, с помощью записи событий, происходящих в работающем ПО и анализа возникающих трасс. Один из инструментов, использующих такие техники — Temporal Rover [279,280].
- *Использующие описание свойств в виде систем переходов или автоматов* (например, [281,282]). В этих техниках трасса анализируется не на выполнение некоторых формул, а на согласованность с заданной автоматной моделью правильного поведения.
- *Использующие программные контракты* (например, [283-285]). В рамках таких подходов корректность поведения системы проверяется во время ее работы с помощью оракулов на базе программных контрактов, внедренных в систему мониторинга.

Более детальный обзор инструментов мониторинга формальных свойств можно найти в [286]. Пока методы мониторинга формальных свойств используются не в исследовательских целях достаточно редко, в основном, для мониторинга небольших критических приложений. Так, NASA применяет несколько таких инструментов, в том числе, разработанный силами собственных сотрудников Java PathExplorer [287], для верификации систем управления спутниковыми системами.

### 3.5.3. Статический анализ формальных свойств

Методы статического анализа формальных свойств исследуются уже несколько десятилетий, но только в последние 7-10 лет появились реализующие их инструменты, пригодные для применения к сложным программным системам.

По используемым этими инструментами техникам формального анализа их можно разделить на следующие группы.

- Методы на основе генерации верификационных утверждений (verification conditions) и их дедуктивного анализа. Подобные методы статического анализа были названы *расширенным статическим анализом* (extended static checking) [288]. На них основаны такие инструменты как ESC/Java 2 [289,290] или Boogie [291], требующие пополнения кода ПО

предусловиями и постусловиями отдельных функций (часто также инвариантами типов данных и циклов), и не требующие вмешательства человека инструменты, такие как Saturn [292] или Calysto [293], нацеленные на поиск типичных ошибок.

- Методы на основе проверки свойств моделей, автоматически создаваемых на основе исходного кода. Такие методы используют так называемую *абстрактную интерпретацию* (abstract interpretation) [294] для построения простых моделей поведения кода, которые позволяют проверить определенные свойства, но могут терять информацию, касающуюся других свойств. Такие методы, в свою очередь, делятся на две группы.
    - Методы на основе булевских моделей. Эти модели являются конечными наборами булевских значений, выбираемыми так, чтобы достаточно точно можно было анализировать возможные сценарии выполнения программы. Такие модели используются в инструменте Blast [295,296] и в SLAM [65,297], который был переработан в Microsoft в Static Driver Verifier [298], используемый для эффективного поиска ошибок, связанных с использованием ресурсов операционной системы в драйверах Microsoft Windows.
    - Методы на основе разнородных моделей. Такие техники используют более сложные модели, чем булевские (например, представление областей возможных значений переменных в виде выпуклых многогранников или деревья выбора значений числовых переменных в зависимости от набора булевских), которые позволяют эффективнее анализировать специфические свойства ПО. К инструментам такого рода относятся ASTREE [299,300] и PolySpace Verifier [103,104].

Хотя инструменты статического анализа формальных свойств появились не так давно, некоторые из них (SLAM в виде Static Driver Verifier) уже используются в промышленном программировании. Стоит обратить внимание, что при практическом использовании основным недостатком инструментов статического анализа становится большое количество сообщений о возможных ошибках или дефектах, которые таковыми не являются. При работе с программными системами размеров в несколько

сотен тысяч строк могут возникать тысячи таких сообщений, проанализировать их вручную в этих случаях не представляется возможным. Поэтому большое количество исследований ведется в направлении создания инструментов, выдающих как можно меньше ложных сообщений об ошибках, и в то же время, не пропускающих серьезные дефекты.

### 3.5.4. Синтетические методы генерации структурных тестов

В последние 3-5 лет активно разрабатываются инструменты автоматической генерации тестов на основе кода, которые используют дополнительные источники информации. В качестве таких источников выступают статический анализ кода, формальный анализ, мониторинг выполнения ранее построенных тестов и т.п. Поскольку в инструментах этого типа используется обычно 3-4 техники разных типов по используемой в этом обзоре классификации, методы, лежащие в их основе, вынесены в отдельную разновидность синтетических методов верификации.

Характерным для этих инструментов примером является развитие инструмента JCrasher [301,302], созданного в 2003-2004 годах в университете Орегона, впоследствии переработанного сначала в Check-n-Crash [303], а затем — в DSDCrasher [304,305].

- JCrasher генерирует структурные тесты для Java-программ, используя случайные данные примитивных типов, несколько простых эвристик нацеливания на вероятные ошибки, синтаксис операций и структуру данных. Получаемый тест представляет собой последовательность вызовов операций, аргументы которых могут быть сгенерированы случайно или получены как результаты предыдущих вызовов. Выполняемые проверки сводятся к отсутствию исключений и сбоев.
- В инструменте Check-n-Crash добавлен предварительный этап статического анализа реализации тестируемых операций, на котором выделяются предикаты, соответствующие различным путям выполнения операций, которые затем разрешаются частично с помощью разрешения ограничений, частично за счет небольших модификаций случайных тестов.
- DSDCrasher добавляет еще одну предварительную фазу, на которой тестируемая программа выполняется на множестве случайных сценариев, и с помощью дополнительного инструмента Daikon [306] выявляются ее возможные

инварианты и ограничения. Затем они используются для отсеивания некорректных сценариев тестирования, которые приводят к ошибкам не в силу ошибочной работы программы, а из-за неправильного ее использования.

- Другие примеры подобной интеграции различных техник верификации в инструментах генерации структурных тестов дают инструменты Randoop [307] и SMART [308]. Они также основаны на случайной генерации последовательностей тестовых вызовов, нацеливаемой на сложные ошибки взаимодействия. В Randoop это нацеливание происходит за счет сокращения множества возможных состояний тестируемой программы при помощи символического анализа выполнения получаемых тестов и использования эвристик попадания в новые ситуации — иногда генерируются длинные последовательности одинаковых вызовов. SMART вместе с символическим выполнением использует выделение возможных ограничений, как и DSDCrasher.

## *Заключение*

Данная работа представляет собой обзор современных методов верификации программного обеспечения. В ней обсуждается место верификации в проектах по разработке и сопровождению ПО, а также различные способы ее выполнения, как широко применяемые в промышленной практике, так и используемые пока только в исследовательских проектах. В круг рассматриваемых методов входят различные виды экспертиз, формальные методы верификации, тестирование и мониторинг, статический анализ. Для каждого разбираемого метода приводится информация о поддерживающих его инструментах. Также дается обзор новых методов верификации, разрабатываемых в последние 5-10 лет и объединяющих элементы формальных методов с другими подходами.

## *Литература*


[1] B. W. Boehm. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall PTR, 1981.

Русский перевод: Б. У. Боэм. *Инженерное проектирование программного обеспечения*. М.: Радо и связь, 1985.

[2] H. Miller, J. Sanders. *Scoping the Global Market: Size Is Just Part of the Story*. IT Professional, 1(2):49-54, 1999.

[3] F. P. Brooks. *No Silver Bullet — Essence and Accidents of Software Engineering*. Proceedings of the IFIP 10-th World Computing Conference, pp. 1069-1076, 1986. Издана по-русски в сборнике Ф. Брукс. *Мифический человеко-месяц, или Как создаются программные системы*. СПб.: Символ-Плюс, 1999.

[4] *The Economic Impacts of Inadequate Infrastructure for Software Testing*. NIST Report, May 2002.

http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[5] http://nssdc.gsfc.nasa.gov/nmc/tmp/MARIN1.html.

[6] N. Levenson, C. S. Turner. *An Investigation of the Therac-25 Accidents*. IEEE Computer, 26(7):18-41, July 1993.

[7] R. Z. Sagdeev, A. V. Zakharov. *Brief history of the Phobos mission*. Nature 341:581-585, 1989.

[8] G. N. Lewis, S. Fetter, L. Gronlund. *Casualties and Damage from Scud Attacks in the 1991 Gulf War*, 1993.

http://web.mit.edu/ssp/Publications/working_papers/wp93-2.pdf.

[9] http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html.

[10] *Mars Climate Orbiter Mishap Investigation Board Phase I Report*, 1999.

ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.

[11] http://www.nyiso.com/public/webdocs/newsroom/press_releases/2005/blackout_rpt_final.pdf.

[12] IEEE 1012-2004 *Standard for Software Verification and Validation*. IEEE, 2005.

[13] IEEE 610.12-1990 *Standard Glossary of Software Engineering Terminology, Corrected Edition*. IEEE, February 1991.

[14] B. W. Boehm. *Software Engineering; R&D Trends and Defense Needs*. In R. Wegner, ed. *Research. Directions in Software Technology*. Cambridge, MA:MIT Press, 1979.

[15] ISO/IEC 12207 *Systems and software engineering — Software life cycle processes.* Geneva, Switzerland: ISO, 2008.

[16] J. McCall, P. Richards, G. Walters. *Factors in Software Quality*. 3 vol., NTIS AD-A049-014, AD-A049-015, AD-A049-055, November 1977.

[17] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. MacLeod, and M. J. Merritt. *Characteristics of Software Quality*. North Holland, 1978.

Русский перевод: Б. Боэм, Дж. Браун, Х. Каспар и др. *Характеристики качества программного обеспечения*. М., Мир, 1991.

[18] G. Murine, C. Carpenter. *Applying Software Quality Metrics*. 1983 ASQC Quality Congress Transactions. Boston, 1983.

[19] L. Arthur. *Measuring Programmer Productivity and Software Quality*. NY: John Wiley & Sons, 1985.

[20] T. Bowen et al. *Specification of Software Quality Attributes*, 3 vol. RADC Report TR-85-37, 1985.

[21] ISO/IEC 9126-1 *Software engineering – Product quality – Part 1: Quality model.* Geneva, Switzerland: ISO, 2001.

[22] ISO/IEC TR 9126-2 *Software engineering – Product quality – Part 2: External metrics.* Geneva, Switzerland: ISO, 2003.

[23] ISO/IEC TR 9126-3 *Software engineering – Product quality – Part 3: Internal metrics.* Geneva, Switzerland: ISO, 2003.

[24] ISO/IEC TR 9126-4 *Software engineering – Product quality – Part 4: Quality in use metrics.* Geneva, Switzerland: ISO, 2004.

[25] ISO/IEC 25000 *Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE.* Geneva, Switzerland: ISO, 2005.

[26] H. D. Benington. *Production of Large Computer Programs*. Proceedings of the ONR Symposium on Advanced Program Methods for Digital Computers, June 1956, pp. 15-27.

Переиздана в Annals of the History of Computing, October 1983, pp. 350-361.

[27] W. W. Royce. *Managing the Development of Large Software Systems*. Proceedings of IEEE WESCON, pp. 1-9, August 1970.
Переиздана в Proceedings of the 9th International Software Engineering Conference, Computer Society Press, pp. 328-338, 1987.

[28] IEEE 1074-2006 *Standard for Developing a Software Project Life Cycle Processes*. IEEE, 2006.

[29] ISO/IEC 15288 *Systems engineering — System life cycle processes*. Geneva, Switzerland: ISO, 2002.

[30] ISO/IEC 15504-1 *Information technology — Process assessment, Part 1: Concepts and vocabulary*. Geneva, Switzerland: ISO, 2004.

[31] IEEE 830-1998. *Recommended Practice for Software Requirements Specifications*. New York: IEEE, 1998.

[32] IEEE 1233-1998. *Guide for Developing System Requirements Specifications*. New York: IEEE, 1998.

[33] IEEE 1059-1993. *Guide for Software Verification and Validation Plans*. New York: IEEE, 1993.

[34] IEEE 829-1998. *Standard for Software Test Documentation*. New York: IEEE, 1998.

[35] IEEE 1008-1987. *Standard for Software Unit Testing*. In IEEE Standards: Software Engineering, Volume Two: Process Standards. New York: IEEE, 1999.

[36] ISO/IEC 14598-1 *Information technology — Software product evaluation — Part 1: General overview*. Geneva, Switzerland: ISO, 1999.

[37] ISO/IEC 14598-2 *Information technology — Software product evaluation — Part 2: Planning and management*. Geneva, Switzerland: ISO, 2000.

[38] ISO/IEC 14598-3 *Information technology — Software product evaluation — Part 3: Process for developers*. Geneva, Switzerland: ISO, 2000.

[39] ISO/IEC 14598-4 *Information technology — Software product evaluation — Part 4: Process for acquirers*. Geneva, Switzerland: ISO, 1999.

[40] ISO/IEC 14598-5 *Information technology — Software product evaluation — Part 5: Process for evaluators*. Geneva, Switzerland: ISO, 1998.

[41] ISO/IEC 14598-6 *Information technology — Software product evaluation — Part 6: Documentation of evaluation modules*. Geneva, Switzerland: ISO, 2001.

[42] ISO/IEC 12119 *Information technology — Software packages — Quality requirements and testing*. Geneva, Switzerland: ISO, 1994.

[43] IEEE 1465 *Adoption of International Standard ISO/IEC 12119:1994 Information Technology — Software Packages — Quality Requirements and Testing*. New York: IEEE, 1998.

[44] ISO/IEC 25051 *Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Requirements for quality of Commercial Off-The-Shelf (COTS) software product and instructions for testing*. Geneva, Switzerland: ISO, 2006.

[45] ISO/IEC 15504-2 *Information technology — Process assessment — Part 2: Performing an assessment*. Geneva, Switzerland: ISO, 2003.

[46] ISO/IEC 15504-3 *Information technology — Process assessment — Part 3: Guidance on performing an assessment*. Geneva, Switzerland: ISO, 2004.

[47] ISO/IEC 15504-4 *Information technology — Process assessment — Part 4: Guidance on use for process improvement and process capability determination*. Geneva, Switzerland: ISO, 2004.

[48] ISO/IEC 15504-5 *Information technology — Process assessment — Part 5: An exemplar Process Assessment Model*. Geneva, Switzerland: ISO, 2006.

[49] http://www.sqi.gu.edu.au/spice/contents.html.

[50] *Capability Maturity Model Integration (CMMI), Version 1.1. CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1). Continuous Representation*. SEI Technical Report CMU/SEI-2002-TR-011, Software Engineering Institute, Pittsburgh, March 2002.
http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr011.pdf.

[51] *Capability Maturity Model Integration (CMMI), Version 1.1. CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1). Staged Representation*. SEI Technical Report CMU/SEI-2002-TR-012, Software Engineering Institute, Pittsburgh, March 2002.
http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr012.pdf.

[52] IEEE 1028 *Standard for Software Reviews*. New York: IEEE, 1998.

[53] IEEE 1044 *Standard Classification for Software Anomalies*. New York: IEEE, 1993.

[54] IEEE 1044.1 *Guide to Classification for Software Anomalies*. New York: IEEE, 1995.

[55] BS 7925-2 *Standard for Software Component Testing*. Working Draft 3.4, British Computer Society, 2001. http://www.testingstandards.co.uk/Component%20Testing.pdf.

[56] B. Boehm, V. Basili. *Software Defect Reduction Top 10 List*. IEEE Computer, 34(1):135-137, January 2001.

[57] L. E. Deimel, S. Rifkin. *Applying Program Comprehension Techniques to Improve Software Inspections*. The Software Practitioner 5(3):4-6, May-June 1995.

[58] T. Gilb, D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[59] A. Porter, H. Siy, L. Votta. *A Review of Software Inspections*. University of Maryland at College Park, Technical Report CS-TR-3552, 1995.

[60] O. Laitenberger. *A Survey of Software Inspection Technologies*. In Handbook on Software Engineering and Knowledge Engineering, v. 2, pp. 517-555. World Scientific Publishing, 2002.

[61] Y. K. Wong. *Modern Software Review: Techniques and Technologies*. IRM Press, 2006.

[62] E. P. Doolan. *Experience with Fagan's inspection method*. Software: Practice & Experience, 22:173-182, 1992.

[63] C. Kaner. *The Performance of the N-Fold Requirement Inspection Method*. Requirements Engineering Journal, 2(2):114-116, 1998.

[64] http://research.microsoft.com/slam/.

[65] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, A. Ustuner. *Thorough Static Analysis of Device Drivers*. In Proc. of EuroSys 2006, ACM SIGOPS Operating Systems Review, 40(4):73-85, October 2006.

[66] http://chacs.nrl.navy.mil/personnel/heitmeyer.html.

[67] C. Heitmeyer, M. Archer, R. Bharadwaj, R. Jeffords. *Tools for constructing requirements specifications: The SCR toolset at the age of ten*. Journal of Computer Systems Science and Engineering, 20(1):19-35, January 2005.

[68] J. Barnes with Praxis Critical Systems Ltd. *High Integrity Software. The Spark Approach to Safety and Security*. Addison-Wesley, 2003.

[69] A. Gupta. *Formal Hardware Verification Methods: A Survey*. Formal Methods in System Design, 1:151-238, 1992.

[70] C. Kern, M. Greenstreet. *Formal Verification in Hardware Design: A Survey*. ACM Transactions on Design Automation of Electronic Systems, 4:123-193, April 1999.

[71] C. Jacob, C. Berg. *Formal Verification of the VAMP Floating Point Unit.* Formal Methods in System Design, 26(3):227-266, Springer Netherlands, 2005.

[72] M. Prasad, A. Biere, A. Gupta. *A Survey of Recent Advances in SAT-Based Formal Verification*. Software Tools for Technology Transfer, 7(2):156-173, 2005.

[73] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner (eds.). *Model Based Testing of Reactive Systems*. LNCS 3472, Springer, 2005.

[74] M. E. Fagan. *Design and Code Inspections to Reduce Errors in Program Development*. IBM Systems Journal, 15(3):182-211, 1976.

[75] M. E. Fagan. *Advances in Software Inspections*. IEEE Transactions on Software Engineering 12(7):744-175, July 1986.

[76] O. Laitenberger, J. M. Debaud. *An encompassing life cycle centric survey of software inspection*. The Journal of Software and Systems, 50(1):5-31, 2000.

[77] L. G. Votta. *Does every inspection need a meeting?* ACM SIGSOFT Software Engineering Notes, 18(5):107-114, 1993.

[78] M. Dyer. *Verification-based Inspection*. Proc. of 26-th Annual Hawaii International Conference on Systems Sciences, pp. 418-427, 1992.

[79] A. A. Porter, L. G. Votta. *An experiment to assess different defect detection methods for software requirements inspections*. Proc. of 16-th International Conference on Software Engineering, pp. 103-112, 1994.

[80] D. V. Bisant, J. B. Lyle. *A two-person inspection method to improve programming productivity*. IEEE Transactions on Software Engineering, 15(10):1294-1304, 1989.

[81] D. L. Parnas, D. Weiss. *Active Design Reviews: Principles and Practices*. Proc. of 8-th International Conference on Software Engineering, pp. 132-136, 1985.

[82] R. N. Britcher. *Using Inspections to Investigate Program Correctness*. IEEE Computer, pp. 38-44, November 1988.

[83] J. C. Knight, E. A. Myers. *Phased Inspections and their Implementation*. ACM SIGSOFT Software Engineering Notes, 16(3):29-35, 1991.

[84] G. M. Schneider, J. Martin, W. T. Tsai. *An experimental study of fault detection in user requirements documents*. ACM Transactions on Software Engineering and Methodology, 1(2):188-204, 1992.

[85] J. Nielsen. *Usability Engineering*. Academic Press, Boston, 1993.

[86] Л. Константайн, Л. Локвуд. *Разработка программного обеспечения*. СПб.: Питер, 2004.

[87] http://www.kb.cert.org/vuls/.

[88] http://nvd.nist.gov/.

[89] B. Schneier. *Secrets and Lies: Digital Security in a Networked World*. John Wiley & Sons, 2000.

Русский перевод: Б. Шнайер. *Секреты и ложь: безопасность данных в цифровом мире*. СПб.: Питер, 2003.

[90] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. N.Y.: John Wiley & Sons, 2001.

[91] L. Dobrica, E. Niemela. *A Survey on Software Architecture Analysis Methods*. IEEE Transactions on Software Engineering, 28(7):638-653, 2002.

[92] M. T. Ionita, D. K. Hammer, H. J. Obbink. *Scenario-Based Software Architecture Evaluation Methods: An Overview*. Technical Note based on the SARA workshop paper presented at ICSE 2002.

[93] M. A. Babar, L. Zhu, R. Jeffery. *A Framework for Classifying and Comparing Software Architecture Evaluation Methods*. Proc. of Australian Software Engineering Conference ASWEC'04, pp. 309-318, 2004.

[94] R. Kazman, L. Bass, G. Abowd, M. Webb. *SAAM: A Method for Analyzing the Properties of Software Architectures*. Proc. of 16-th International Conference on Software Engineering, pp. 81-90, 1994.

[95] R. Kazman, G. Abowd, L. Bass, P. Clements. *Scenario-Based Analysis of Software Architecture*. IEEE Software, 13(6):47-55, November 1996.

[96] R. Kazman, M. Klein, M. Barbacci, H. Lipson, T. Longstaff, S. J. Carriere. *The Architecture Tradeoff Analysis Method*. Proc. of 4-th International Conference on Engineering. of Complex Computer Systems (ICECCS '98), pp. 68-78, August 1998.

[97] R. Kazman, M. Barbacci, M. Klein, S. J. Carriere, S. G. Woods. *Experience with Performing Architecture Tradeoff Analysis.* Proc. of International Conference of Software Engineering (ICSE'99), pp. 54-63, May 1999.

[98] P. O. Bengtsson, J. Bosch. *Scenario-Based Architecture Reengineering*. Proc. Of 5-th International Conference on Software Reuse (ICSR 5), pp. 308-317, 1998.

[99] P. O. Bengtsson, N. Lassing, J. Bosch, H. V. Vliet. *Architecture-Level Modifiability Analysis*. Journal of Systems and Software, 69(1-2):129-147, January 2004.

[100] R. Kazman, J. Asundi, M. Klein. *Quantifying the Costs and Benefits of Architectural Decisions*. Proc. of 23-rd International Conference on Software Engineering (ICSE'01), pp. 297-306, 2001.

[101] B. Lionberger, C. Zhang. *ATAM Assistant: A Semi-Automated Tool for the Architecture Tradeoff Analysis Method*. Proc. of Software Engineering and Applications Conference, 2007.

[102] http://www.testingfaqs.org/t-static.html.

[103] P. Emanuelsson, U. Nilsson. *A Comparative Study of Industrial Static Analysis Tools*. Technical Report 2008:3, Linkoping University, 2008. http://www.ep.liu.se/ea/trcis/2008/003/trcis08003.pdf.

[104] http://www.mathworks.com/products/polyspace/index.html?s_cid=psr_prod.

[105] http://www.coverity.com/html/prod_prevent.html.

[106] http://www.iplbath.com/pdf/klocwork/KlocworkEmbeddedSystems.pdf.

[107] L. Yu, R. B. France, I. Ray, K. Lano. *A light-weight static approach to analyzing UML behavioral properties*. Proc. of 12-th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007), pp. 56-63, 2007.

[108] Э. Мендельсон. *Введение в математическую логику*. М.: Наука, 1971.

[109] Ю. Л. Ершов, Е. А. Палютин. *Математическая логика*. СПб.: Лань, 2004.

[110] Х. Барендрегт. *Лямбда-исчисление. Его синтаксис и семантика*. М.: Мир, 1985.

[111] J.-F. Monin. *Understanding Formal Methods*. Springer, 2003.

[112] Р. Фейс. *Модальная логика*. М.: Наука, 1974.

[113] К. Ии, Н. В. Шилов, Е. В. Бодин. *О программных логиках — просто*. Системная Информатика, 8:206-249, Новосибирск: Наука, 2002.

[114] Э. Кларк, О. Грамберг, Д. Пелед. *Верификация моделей программ: Model Checking*. М.: МЦНМО, 2002.

[115] К. Дж. Дейт. *Введение в системы баз данных*. 8-е изд. М.: Вильямс, 2006.

[116] Г. Гарсиа-Молина, Дж. Ульман, Дж. Уидом. *Системы баз данных. Полный курс*. М.: Вильямс, 2003.

[117] В. Е. Котов, Л. А. Черкасова. *Исчисления процессов I*. Системная информатика, 2:6-38, Новосибирск: Наука, 1993.

[118] C. A. R. Hoare. *Communicating sequential processes*. Prentice Hall, 1985.
Русский перевод: Ч. Хоар. *Взаимодействующие последовательные процессы*. М.: Мир, 1989.

[119] R. Milner. *A Calculus of Communicating Systems*. LNCS 92, Springer-Verlag, 1980.

[120] J. A. Bergstra, J. W. Klop. *Fixed point semantics in process algebra*. Technical Report IW 208, Mathematical Centre, Amsterdam, 1982.

[121] J. C. M. Baeten, W. P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science, 18. Cambridge University Press, 1990.

[122] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[123] L. Cardelli, A. D. Gordon. *Mobile Ambients*. Proc. of 1-st International Conference on Foundations of Software Science and Computation Structure, M. Nivat, ed. LNCS 1378:140-155, Springer-Verlag, 1998.

[124] Дж. Э. Хопкрофт, Р. Мотвани, Дж. Д. Ульман. *Введение в теорию автоматов, языков и вычислений*. 2-е изд. М.: Вильямс, 2002.

[125] В. Б. Кудрявцев, С. В. Алешин, А. С. Подколзин. *Введение в теорию конечных автоматов*. М.: Наука, 1985.

[126] Е. В. Кузьмин, В. А. Соколов. *Структурированные системы переходов*. М.: ФИЗМАТЛИТ, 2006.

[127] G. A. Simon. *An extended finite state machine approach to protocol specification*. Proc. of 2-nd International Workshop on Protocol Specification, Testing and Verification, pp. 113-133, North-Holland Publishing Co., 1982.

[128] D. L. Dill. *Timing assumptions and verification of finite-state concurrent systems*. In J. Sifakis, ed. Proc. of International Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407:197-212, Springer, 1989.

[129] R. Alur, D. L. Dill. *A theory of timed automata*. Theoretical Computer Science 126:183-235, 1994.

[130] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, S. Yovine. *The algorithmic analysis of hybrid systems*. Theoretical Computer Science, 138(1):3-34,1995.

[131] T. A. Henzinger. *The Theory of Hybrid Automata*. Proc. of 11-th Annual IEEE Symposium on Logic in Computer Science (LICS 1996), pp. 278-292, 1996.

[132] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut fuer Instrumentelle Mathematik, Bonn, 1962.

[133] Дж. Питерсон. *Теория сетей Петри и моделирование систем*. М.: Мир, 1984.

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

[135] W. Thomas. *Automata on infinite objects*. In J. van Leeuwen, ed. Handbook of Theoretical Computer Science, vol. B, pp. 133-191. 1990.

[136] D. Harel. *Statecharts: A visual formalism for complex systems*. Science of Computer Programming, 8(3):231-274, June 1987.

[137] Y. Gurevich. *Evolving Algebras: An Introductory Tutorial*. Bulletin of the European Association for Theoretical Computer Science 43:264-284, February 1991.

[138] Ю. Гуревич. *Последовательные машины абстрактных состояний охватывают последовательные алгоритмы*. Системная информатика, 9:7-50, Новосибирск: Изд-во СО РАН, 2004.

[139] C. A. R. Hoare. *An axiomatic basis for computer programming*. Communications of the ACM, 12(10):576–585, October 1969.

[140] V. R. Pratt. *Semantical Considerations on Floyd-Hoare Logic*. Proc. of 17-th Annual IEEE Symposium on Foundations of Computer Science, pp. 109-121, 1976.

[141] D. Harel, D. Kozen, J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[142] B. Meyer. *Design by Contract*. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[143] B. Meyer. *Applying "Design by Contract"*. IEEE Computer 25(10):40-51, October 1992.

[144] R. W. Floyd. *Assigning meaning to programs*. Proc. of Symposium in Applied Mathematics. J. T. Schwartz, ed. Mathematical Aspects of Computer Science, 19:19-32, 1967.

[145] E. A. Dijkstra. *Discipline of programming*. Prentice Hall, 1976.
Русский перевод: Э. Дейкстра. Дисциплина программирования. М.: Мир, 1978.

[146] S. Owicki, D. Gries. *Verifying properties of parallel programs: an axiomatic approach*. Communications of the ACM, 19(5):279-285, May 1976.

[147] Ч. Чень, Р. Ли Математическая логика и автоматическое доказательство теорем. М.: Мир, 1973.

[148] http://en.wikipedia.org/wiki/Automated_theorem_proving.

[149] http://www.cs.utexas.edu/~moore/acl2/.

[150] M. Kaufmann, J. S. Moore. *Design Goals for ACL2*. Proc. of 3-rd International School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, pp. 92-117, 1994.

[151] http://www.eprover.org/.

[152] L. Bachmair, H. Ganzinger. *Rewrite-Based Equational Theorem Proving with Selection and Simplification*. Journal of Logic and Computation, 4(3):217-247, 1994.

[153] http://www4.informatik.tu-muenchen.de/~schulz/WORK/e-setheo.html.

[154] http://www.key-project.org/.

[155] B. Beckert, R. Hähnle, P. H. Schmitt, eds. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.

[156] A. Riazanov, A. Voronkov. *The Design and Implementation of Vampire*. AI Communications, 15(2):91-110, September 2002.

[157] http://www.mpi-inf.mpg.de/~hillen/waldmeister/.

[158] http://combination.cs.uiowa.edu/Darwin/.

[159] http://www.cs.miami.edu/~tptp/CASC/.

[160] E. Denney, B. Fischer, J. Schumann. *An Empirical Evaluation of Automated Theorem Provers in Software Certification*. International Journal on Artificial Intelligence Tools, 15(1):81-107, 2006.

[161] http://pvs.csl.sri.com/.

[162] S. Owre, N. Shankar, J. Rushby. *PVS: A Prototype Verification System*. Proc. of 11-th International Conference on Automated Deduction, LNCS 607:748-752, Springer, June 1992.

[163] http://hol.sourceforge.net/.

[164] M. Gordon. *HOL: A machine oriented formulation of higher order logic*. Technical Report 68, Computer Laboratory, University of Cambridge, July 1985.

[165] http://isabelle.in.tum.de/.

[166] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283, Springer, 2002

[167] http://coq.inria.fr/.

[168] Y. Bertot. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[169] http://hol.sourceforge.net/hol-biblio.html.

[170] J. E. M. Clarke, E. A. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Proc. of Workshop on Logic of Programs, LNCS 131:52-71, Springer, 1981.

[171] M. Ben-Ari, Z. Manna, A. Pnueli. *The temporal logic of branching time*. Proc. of 8-th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 164-176, January 1981.

[172] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill. *Sequential circuit verification using symbolic model checking*. Proc. of 27-th ACM/IEEE Design Automation Conference, pp. 46-51, June 1990.

[173] A. Pnueli. *The temporal logic of programs*. Proc. of 18-th IEEE Symposium on Foundations of Computer Science, pp. 46-67, 1977.

[174] E. A. Emerson, J. Y. Halpern. *"Sometimes" and "not never" revisited: On branching versus linear time*. Journal of the ACM, 33(1):151–178, 1986.

[175] A. Browne, E. M. Clarke, S. Jha, D. E. Long, W. Marrero. *An improved algorithm for the evaluation of fixpoint expressions*. Theoretical Computer Science, 178(1):237-255, May 1997.

[176] http://spinroot.com/spin/whatispin.html.

[177] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[178] R. Mateescu, M. Sighireanu. *Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus*. Science of Computer Programming, 46(3):255-281, 2003.

[179] http://www.inrialpes.fr/vasy/cadp/index.html.

[180] http://www.cs.cmu.edu/~modelcheck/smv.html.

[181] http://nusmv.fbk.eu/.

[182] S. Christensen, J. B. Jørgensen, L. M. Kristensen. *Design/CPN — A computer tool for Coloured Petri Nets*. Proc. of TACAS'1997, LNCS 1217:209-223, Springer, 1997.

[183] http://www.uppaal.com/.

[184] http://www-verimag.imag.fr/TEMPORISE/kronos/.

[185] http://bogor.projects.cis.ksu.edu/.

[186] http://embedded.eecs.berkeley.edu/research/hytech/.

[187] http://www.veritable.com/verity-check.html.

[188] http://en.wikipedia.org/wiki/List_of_performance_analysis_tools.

[189] М. А. Посыпкин, А. А. Соколов. *Обзор методов автоматизации мониторинга, анализа и визуализации поведения параллельных процессов, взаимодействующих с помощью передачи сообщений*. Препринт ИСП РАН 7, М.: ИСП РАН, 2005.

[190] http://www.monitortools.com/security/.

[191] http://www.windowsecurity.com/software/Network-Auditing/.

[192] http://www.gnu.org/software/binutils/.

[193] http://gcc.gnu.org/.

[194] http://msdn.microsoft.com/en-us/library/z9z62c29(VS.80).aspx.

[195] http://valgrind.org/.

[196] N. Nethercote, J. Seward. *Valgrind: a framework for heavyweight dynamic binary instrumentation*. Proc. of 2007 PLDI conference, ACM SIGPLAN Notices, 42(6):89-100, June 2007.

[197] http://www-306.ibm.com/software/awdtools/purifyplus/.

[198] http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm.

[199] ISO/IEC TR 19759 *Software Engineering — Guide to the Software Engineering Body of Knowledge (SWEBOK)*. Geneva, Switzerland: ISO, 2005.

[200] H. Zhu, P. A. V. Hall, J. H. R. May. *Software Unit Test Coverage and Adequacy*. ACM Computing Surveys, 29(4):366-427, Dec. 1997.

[201] B. Beizer. *Software Testing Techniques*. International Thomson Press, 1990.

[202] A. P. Mathur. *Foundations of Software Testing*. Copymat Services, 2006.

[203] G. Vijayaraghavan, C. Kaner. *Bug Taxonomies: Use Them to Generate Better Tests*. STAREAST, 2003. http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=6982.

[204] R. G. Hamlet. *Testing programs with the aid of a compiler*. IEEE Transactions on Software Engineering, 3(4):279–290, July 1977.

[205] J. Offutt, J. Voas, J. Payne. *Mutation Operators for Ada*. Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University, October 1996.

[206] Y.-S. Ma, Y.-R. Kwon, J. Offutt. *Inter-Class Mutation Operators for Java*. Proc. of 13-th International Symposium on Software Reliability Engineering, pp. 352-363, November 2002.

[207] A. S. Namin, J. H. Andrews, D. J. Murdoch. *Sufficient mutation operators for measuring test effectiveness*. Proc. of 30-th International Conference on Software Engineering, pp. 351-360, 2008.

[208] D. Hamlet. *Random Testing*. In J. Marciniak, ed. Encyclopedia of Software Engineering, pp. 970-978, Wiley, 1994.

[209] А. А. Марков. *Исследование замечательного случая зависимых испытаний*. Известия Императорской Академии наук, серия VI, 1(3), 1907.

[210] J. G. Kemeny, J. L. Snell. *Finite Markov Chains*. Springer-Verlag, 1976.
Русский перевод: Дж. Кемени, Дж. Снелл. *Конечные цепи Маркова*. М.: Наука, 1970.

[211] T. J. Ostrand, M. J. Balcer. *The Category-Partition Method for Specifying and Generating Functional Tests*. Communications of the ACM, 31(3):676-686, June 1988.

[212] В. В. Липаев. *Тестирование программ*. М: Радио и связь, 1986.

[213] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.
Русский перевод: Майерс Г. *Искусство тестирования программ*. М.: Финансы и статистика, 1982.

[214] M. Grindal, J. Offutt, S. Andler. *Combination Testing Strategies: A Survey*. Software Testing, Verification & Reliability, 15(3):167-199, 2005.

[215] J. Ryser, M. Glinz. *A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts*. Proc. of 12-th International Conference on Software and Systems Engineering and Their Applications (ICSSEA '99), 1999.

[216] J. Bach. *Exploratory Testing Explained*. 2002. http://www.satisfice.com/articles/et-article.pdf.

[217] R. P. Pargas, M. J. Harrold, R. Peck. *Test-data Generation Using Genetic Algorithms*. Software Testing, Verification & Reliability, 9(4):263–282, 1999.

[218] L. Baresi, M. Young. *Test oracles*. Technical Report CIS-TR-01-02, University of Oregon, Department of Computer and Information Science, Eugene, Oregon, 2001.

[219] http://www.aptest.com/resources.html#mngt.

[220] http://www-306.ibm.com/software/awdtools/test/manager/.

[221] https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1131_4000_100__.

[222] http://www.testingfaqs.org/t-eval.html.

[223] http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks.

[224] P. Hamill. *Unit Test Frameworks*. O'Reilly, 2004.

[225] http://tetworks.opengroup.org/Products/tetware.htm.

[226] http://www.pairwise.org/tools.asp.

[227] C. J. Colbourn. *Combinatorial aspects of covering arrays*. Le Matematiche (Catania), 58:121–167, 2004.

[228] A. Hartman, L. Raskin. *Problems and algorithms for covering arrays*. Discrete Math., 284(1-3):149–156, Jul. 2004.

[229] http://www.aptest.com/resources.html#app-data.

[230] http://www.sqlmanager.net/products.

[231] http://www.sqledit.com/dg/.

[232] http://www.forsql.com/.

[233] D. Barbosa, A. Mendelzon. *Declarative generation of synthetic XML data*. Software: Practice & Experience, 36(10):1051-1079, August 2006.

[234] R. Lämmel, W. Schulte. *Controllable combinatorial coverage in grammar-based testing*. Proc. of TESTCOM 2006, LNCS 3964:19-38, Springer, 2006.

[235] http://www.alphaworks.ibm.com/tech/xmlgenerator.

[236] http://xml-xig.sourceforge.net/.

[237] http://iwm.uni-koblenz.de/datagen/.

[238] P. Purdom. *A sentence generator for testing parsers*. BIT, 12(3):366–375, 1972.

[239] http://www.mmsindia.com/JSynTest.html.

[240] С.В. Зеленов, С.А. Зеленова. *Генерация позитивных и негативных тестов парсеров*. Программирование, 31(6):25–40, 2005.

[241] С. В. Зеленов, С. А. Зеленова, А. С. Косачев, А. К. Петренко. *Генерация тестов для компиляторов и других текстовых процессоров*. Программирование, 29(2):59–69, 2003.

[242] А. В. Демаков, С. В. Зеленов, С. А. Зеленова. *Генерация тестовых данных сложной структуры с учетом контекстных ограничений*. Труды ИСП РАН, 9:83–96, 2006.

[243] http://www.testingfaqs.org/t-gui.html.

[244] http://www.softwareqatest.com/qatweb1.html#FUNC.

[245] http://www-306.ibm.com/software/awdtools/tester/robot/index.html.

[246] https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1352_4000_100__.

[247] http://www.empirix.com/products-services/w-testing-etest.asp.

[248] http://www.aptest.com/resources.html#comm.

[249] E. F. Moore. *Gedanken-experiments on Sequential Machines*. Automata Studies, Annals of Mathematical Studies, 34:129–153. N.J.: Princeton University Press, 1956.

Русский перевод: Э. Ф. Мур *Умозрительные эксперименты с последовательностными машинами*. В книге Автоматы, стр. 179-210. М.: ИЛ, 1956.

[250] F. C. Hennie. *Fault-detecting experiments for sequential circuits*. Proc. of 5-th Annual Symposium on Switching Theory and Logical Design, pp. 95-110, November 1964.

[251] М. П. Василевский. *О распознавании неисправностей автоматов*. Кибернетика, 4:98-108, Киев, 1973.

[252] ISO 9646. *Information Theory — Open System Interconnection — Conformance Testing Methodology and Framework.* ISO, Geneve, 1991.

[253] А. К. Петренко. *Спецификация тестов на основе описания трасс.* Программирование, 19(1):66-73, 1993.

[254] I. Bourdonov, A. Kossatchev, A. Petrenko, D. Galter. *KVEST: Automated Generation of Test Suites from Formal Specifications*. Proc. of Formal Methods'99, LNCS 1708,:608-621, Springer-Verlag, 1999.

[255] M. Utting, B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.

[256] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, M. Utting. *BZ-TT: A tool-set for test generation from Z and B using constraint logic programming*. Proc. of FATES'2002, pp. 105-119, August 2002.

[257] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[258] E. Farchi, A. Hartman, S. S. Pinter. *Using a model-based test generator to test for standard conformance*. IBM Systems Journal, 41(1):89-110, 2002.

[259] D. L. Dill, S. Park, A. G. Nowatzyk. *Formal Specification of Abstract Memory Models*. In G. Borriello, C. Ebeling, eds.: Proc. of the Symposium on Research on Integrated Systems, pp. 38-52, MIT Press, 1993.

[260] http://www.agedis.de/.

[261] E. Brinksma. *A theory for the derivation of tests*. Proc. of 8-th International Conference on Protocol Specification, Testing and Verification, pp. 63-74, North-Holland, 1988.

[262] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[263] J. Tretmans, A. Belinfante. *Automatic testing with formal methods*. Proc. of 7-th European Conference on Software Testing, Analysis and Review, Barcelona, Spain, November 1999.

[264] J.-C. Fernandez, C. Jard, T. Jeron, C. Viho. *Using On-the-Fly Verification Techniques for the Generation of Test Suites*. Proc. of 8-th International Conference on Computer Aided Verification, LNCS 1102:348-359, Springer, 1996.

[265] http://www.unitesk.ru.

[266] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. *Подход UniTesK к разработке тестов*. Программирование, 29(6):25-43, 2003.

[267] http://research.microsoft.com/specexplorer/.

[268] A. Hartman. *Model Based Test Generation Tools*. AGEDIS Consortium, 2002. http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf.

[269] P. Ammann, P. E. Black. *Abstracting formal specifications to generate software tests via model checking*. Proc. of 18-th Digital Avionics Systems Conference, 2:10.A.6-1-10.A.6-10, IEEE, October 1999.

[270] A. Gargantini, C. Heitmeyer. *Using model checking to generate tests from requirements specifications*. ACM SIGSOFT Software Engineering Notes, 24(6):146-162, November 1999.

[271] W. Visser, C. S. Pasareanu, S. Khurshid. *Test input generation with Java PathFinder*. ACM SIGSOFT Software Engineering Notes, 29(4):97-107, July 2004.

[272] C. Engel, R. Hahnle. *Generating unit tests from formal proofs*. Y. Gurevich, B. Meyer, eds. Proc. of TAP 2007. LNCS 4454:169–188, Springer-Verlag, 2007.

[273] A. Gotlieb, B. Botella, M. Rueher. *Automatic test data generation using constraint solving techniques*. ACM SIGSOFT Software Engineering Notes, 23(2):53-62, 1998.

[274] C. Boyapati, S. Khurshid, D. Marinov. *Korat: automated testing based on Java predicates*. Proc. of International Symposium on Software Testing and Analysis, pp. 123–133, 2002.

[275] K. Sen, G. Agha. *CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools*. Proc. of Computer Aided Verification, pp.419-423, August 2006.

[276] M. Utting, A. Pretschner, B. Legeard. *A Taxonomy of Model-Based Testing*. Technical Report, Department of Computer Science, The University of Waikato, New Zealand, 2006.

[277] I. Lee, S. Kannan, M. Kim, O. Sokolsky, M. Viswanathan. *Runtime Assurance Based On Formal Specifications*. Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'1999, pp. 279-287, 1999.

[278] K. Havelund, G. Rosu. *Monitoring Java Programs with Java PathExplorer*. Electronic Notes in Theoretical Computer Science, 55(2):1-18, January 2004.

[279] http://www.time-rover.com.

[280] D. Drusinsky. *Temporal Rover and ATG Rover*. Proc. of 7-th International SPIN Workshop on SPIN Model Checking and Software Verification, LNCS 1885:323-330, Springer-Verlag, 2000.

[281] A. Cavalli, C. Gervy, S. Prokopenko. *New approaches for passive testing using an Extended Finite State Machine specification*. Information and Software Technology, 45(12):837-852, Elsevier, September 2003.

[282] D. Chen, J. Wu, H. Chi. *Passive testing on TCP*. Proc. of International Conference on Communication Technology ICCT 2003, 1:182-186, April 2003.

[283] Y. Cheon, G. T. Leavens. *A runtime assertion checker for the Java Modeling Language (JML)*. Proc. of International Conference on Software Engineering Research and Practice (SERP'02), pp. 322-328, CSREA Press, June 2002.

[284] B. Schoeller. *Strengthening Eiffel Contracts using Models*. Proc. of Workshop on Formal Aspects of Component Software FACS'03, pp. 143-158, September 2003.

[285] M. Barnett, W. Schulte. *Runtime verification of .NET contracts*. Journal of Systems and Software, 65(3):199-208, March 2003.

[286] N. Delgado, A. Q. Gates, S. Roach. *A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools*. IEEE Transactions on Software Engineering, 30(12):859-872, December 2004.

[287] K. Havelund, G. Rosu. *An Overview of the Runtime Verification Tool Java PathExplorer*. Formal Methods in System Design, 24(2):189-215, March 2004.

[288] D. L. Detlefs, K. R. M. Leino, G. Nelson, J. B. Saxe. *Extended static checking*. Technical Report SRC-RR-159, Digital Equipment Corporation, Systems Research Center, 1998. Now available from HP Labs.

[289] http://kind.ucd.ie/products/opensource/ESCJava2/.

[290] D. R. Cok, J. R. Kiniry. *ESC/Java2: Uniting ESC/Java and JML*. Proc. of International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04), LNCS 3362:108-128, Springer-Verlag, January 2005.

[291] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, K. R. M. Leino. *Boogie: A modular reusable verifier for object-oriented programs*. Proc. of 4-th International Symposium on Formal Methods for Components and Objects (FMCO 2005), LNCS 4111:364-387, Springer-Verlag, 2006.

[292] Y. Xie, A. Aiken. *Scalable error detection using boolean satisfiability*. ACM SIGPLAN Notices, 40(1):351–363, ACM Press, January 2005.

[293] D. Babic, A. J. Hu. *Calysto: scalable and precise extended static checking*. Proc. of 30-th International conference on Software Engineering, pp. 211-220, 2008.

[294] P. Cousot, R. Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. Proc. of 4-th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (PLoP), pp. 238-252, 1977.

[295] http://mtc.epfl.ch/software-tools/blast/.

[296] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. *Software Verification with Blast*. Proc. of 10-th SPIN Workshop on Model Checking Software (SPIN 2003), LNCS 2648:235-239, Springer-Verlag, 2003.

[297] T. Ball, S. K. Rajamani. *Automatically Validating Temporal Safety Properties of Interfaces*. Proc. of Model Checking of Software, LNCS 2057:103-122, Springer, 2001.

[298] http://www.microsoft.com/whdc/devtools/tools/SDV.mspx.

[299] http://www.astree.ens.fr/.

[300] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, X. Rival. *Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software*. T. Mogensen, D. A. Schmidt, I. H. Sudborough, eds. The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. LNCS 2566:85-108, Springer-Verlag 2002.

[301] http://www.cc.gatech.edu/jcrasher/.

[302] C. Csallner, Y. Smaragdakis. *JCrasher: and Automatic Robustness Tester for Java*. Software — Practice & Experience, 34(11):1025-1050, 2004.

[303] C. Csallner, Y. Smaragdakis. *Check 'n' Crash: Combining static checking and testing*. Proc. of 27-th International Conference on Software Engineering (ICSE), pp. 422-431, ACM, May 2005.

[304] C. Csallner, Y. Smaragdakis. *DSD-Crasher: A hybrid analysis tool for bug finding*. Proc. of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 245-254. ACM, July 2006.

[305] Y. Smaragdakis, C. Csallner. *Combining Static and Dynamic Reasoning for Bug Detection*. Proc. of TAP 2007, LNCS 4454:1-16, Springer, 2007.

[306] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin. *Dynamically discovering likely program invariants to support program evolution*. IEEE Transactions on Software Engineering, 27(2):99-123, February 2001.

[307] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball. *Feedback-Directed Random Test Generation*. Proc. of International Conference on Software Engineering, pp. 75-84, 2007.

[308] P. Godefroid. *Compositional dynamic test generation*. Proc. of 34-th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (PLOP 2007), pp. 47-54, 2007